Search Trees: BSTs and B-Trees

David Kauchak
cs302
Spring 2012

Administrative

- HW grading

Number guessing game

- I'm thinking of a number between 1 and n
- You are trying to guess the answer
- For each guess, I'll tell you "correct", "higher" or "lower"
- Describe an algorithm that minimizes the number of guesses

Binary Search Trees

- BST – A binary tree where a parent's value is greater than all values in the left subtree and less than or equal to all the values in the right subtree

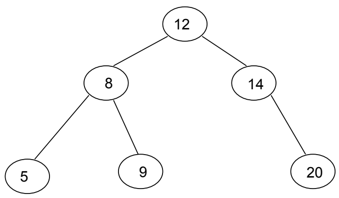
$$leftTree(i) < i \leq rightTree(i)$$

- the left and right children are also binary trees
- Why not?

$$leftTree(i) \leq i \leq rightTree(i)$$

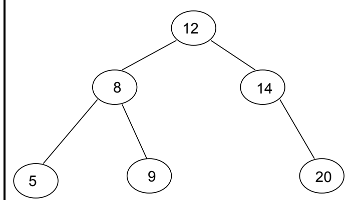
- Can be implemented with with pointers or an array

Example



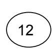
What else can we say?

$$left(i) < i \le right(i)$$

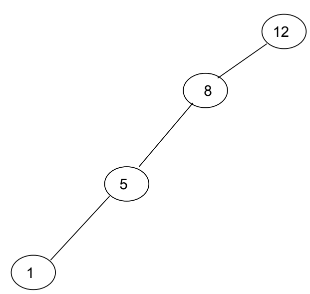


- All elements to the left of a node are less than the node
- All elements to the right of a node are greater than or equal to the node
- The smallest element is the left-most element
- The largest element is the right-most element

Another example: the loner



Another example: the twig



Operations

- Search(T,k) – Does value k exist in tree T
- Insert(T,k) – Insert value k into tree T
- Delete(T,x) – Delete node x from tree T
- Minimum(T) – What is the smallest value in the tree?
- Maximum(T) – What is the largest value in the tree?
- Successor(T,x) – What is the next element in sorted order after x
- Predecessor(T,x) – What is the previous element in sorted order of x
- Median(T) – return the median of the values in tree T

Search

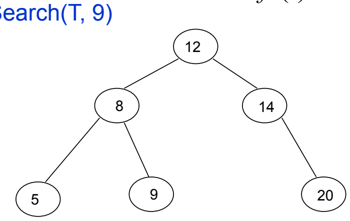
- How do we find an element?

```
BSTSearch(x, k)
  if x = null or k = x
          return x
  elseif k < x
          return BSTSearch(Left(x), k)
  else
          return BSTSearch(Right(x), k)
```

Finding an element

$left(i) < i \leq right(i)$

Search(T, 9)

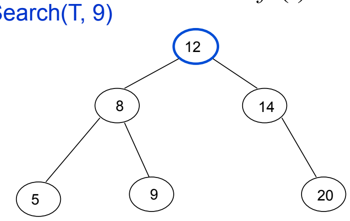



```
BSTSearch(x, k)
  if x = null or k = x
          return x
  elseif k < x
          return BSTSearch(Left(x), k)
  else
          return BSTSearch(Right(x), k)
```

Finding an element

$left(i) < i \leq right(i)$

Search(T, 9)

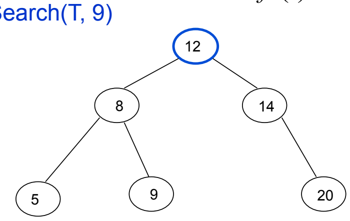



```
BSTSearch(x, k)
  if x = null or k = x
          return x
  elseif k < x
          return BSTSearch(Left(x), k)
  else
          return BSTSearch(Right(x), k)
```

Finding an element

$left(i) < i \le right(i)$

Search(T, 9)



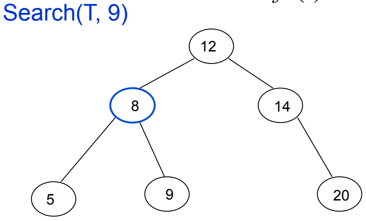
9 > 12?

```
BSTSearch(x, k)
if x = null or k = x
        return x
elseif k < x
        return BSTSearch(Left(x), k)
else
        return BSTSearch(Right(x), k)
```

Finding an element

$left(i) < i \le right(i)$

Search(T, 9)

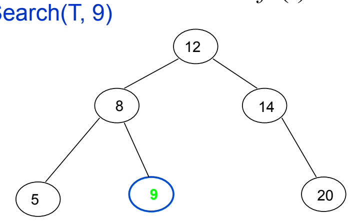



```
BSTSearch(x, k)
if x = null or k = x
        return x
elseif k < x
        return BSTSearch(Left(x), k)
else
        return BSTSearch(Right(x), k)
```

Finding an element

$left(i) < i \le right(i)$

Search(T, 9)

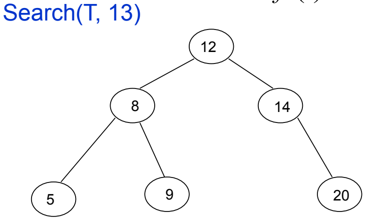



```
BSTSearch(x, k)
if x = null or k = x
        return x
elseif k < x
        return BSTSearch(Left(x), k)
else
        return BSTSearch(Right(x), k)
```

Finding an element

$left(i) < i \le right(i)$

Search(T, 13)

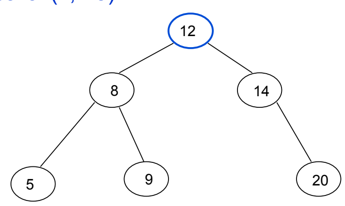



```
BSTSearch(x, k)
if x = null or k = x
        return x
elseif k < x
        return BSTSearch(Left(x), k)
else
        return BSTSearch(Right(x), k)
```

Finding an element

$left(i) < i \le right(i)$

Search(T, 13)

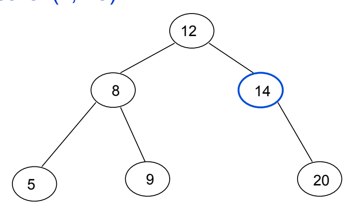



```
BSTSearch(x, k)
if x = null or k = x
        return x
elseif k < x
        return BSTSearch(Left(x), k)
else
        return BSTSearch(Right(x), k)
```

Finding an element

$left(i) < i \le right(i)$

Search(T, 13)

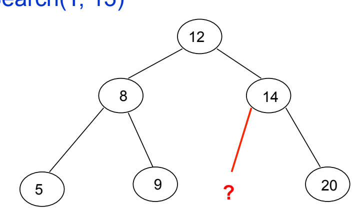



```
BSTSearch(x, k)
if x = null or k = x
        return x
elseif k < x
        return BSTSearch(Left(x), k)
else
        return BSTSearch(Right(x), k)
```

Finding an element

$left(i) < i \le right(i)$

Search(T, 13)




```
BSTSearch(x, k)
if x = null or k = x
        return x
elseif k < x
        return BSTSearch(Left(x), k)
else
        return BSTSearch(Right(x), k)
```

Iterative search

```
IterativeBSTSearch(x, k)
while x ≠ null and k ≠ x
        if k < x
                x ← Left(x)
        else
                x ← Right(x)
return x
```

```
BSTSearch(x, k)
if x = null or k = x
        return x
elseif k < x
        return BSTSearch(Left(x), k)
else
        return BSTSearch(Right(x), k)
```

Is BSTSearch correct?

```
BSTSearch(x, k)
if x = null or k = x
        return x
elseif k < x
        return BSTSearch(Left(x), k)
else
        return BSTSearch(Right(x), k)
```

$$left(i) < i \leq right(i)$$

Running time of BST

- Worst case?
  - O(height of the tree)
- Average case?
  - O(height of the tree)
- Best case?
  - O(1)

Height of the tree

- Worst case height?
  - n-1
  - "the twig"
- Best case height?
  - floor($\log_2 n$)
  - complete (or near complete) binary tree
- Average case height?
  - Depends on two things:
    - the data
    - how we build the tree!

Insertion

```
BSTInsert(T, x)
if Root(T) = null
        Root(T) ← x
else
        y ← Root(T)
        while y ≠ null
                prev ← y
                if x < y
                        y ← Left(y)
                else
                        y ← Right(y)
        Parent(x) ← prev
        if x < prev
                Left(prev) ← x
        else
                Right(prev) ← x
```

Insertion

```
BSTInsert(T, x)
if Root(T) = null
    Root(T) ← x
else
    y ← Root(T)
    while y ≠ null
        prev ← y
        if x < y
            y ← Left(y)
        else
            y ← Right(y)
    Parent(x) ← prev
    if x < prev
        Left(prev) ← x
    else
        Right(prev) ← x
```

Similar to search

```
IterativeBSTSearch(x, k)
while x ≠ null and k ≠ x
    if k < x
        x ← Left(x)
    else
        x ← Right(x)
return x
```

Insertion

```
BSTInsert(T, x)
if Root(T) = null
    Root(T) ← x
else
    y ← Root(T)
    while y ≠ null
        prev ← y
        if x < y
            y ← Left(y)
        else
            y ← Right(y)
    Parent(x) ← prev
    if x < prev
        Left(prev) ← x
    else
        Right(prev) ← x
```

Similar to search

Find the correct location in the tree

Insertion

```
BSTInsert(T, x)
if Root(T) = null
    Root(T) ← x
else
    y ← Root(T)
    while y ≠ null
        prev ← y
        if x < y
            y ← Left(y)
        else
            y ← Right(y)
    Parent(x) ← prev
    if x < prev
        Left(prev) ← x
    else
        Right(prev) ← x
```

keeps track of the previous node we visited so when we fall off the tree, we know

Insertion

```
BSTInsert(T, x)
if Root(T) = null
    Root(T) ← x
else
    y ← Root(T)
    while y ≠ null
        prev ← y
        if x < y
            y ← Left(y)
        else
            y ← Right(y)
    Parent(x) ← prev
    if x < prev
        Left(prev) ← x
    else
        Right(prev) ← x
```

add node onto the bottom of the tree

Correctness?

```
BSTInsert(T, x)
if Root(T) = null
        Root(T) ← x
else
        y ← Root(T)
        while y ≠ null
                prev ← y
                if x < y
                        y ← Left(y)
                else
                        y ← Right(y)
        Parent(x) ← prev
        if x < prev
                Left(prev) ← x
        else
                Right(prev) ← x
```

maintain BST property

Correctness

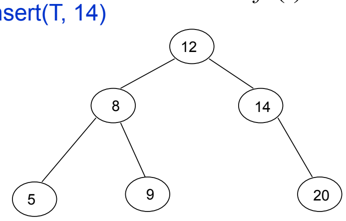
```
BSTInsert(T, x)
if Root(T) = null
        Root(T) ← x
else
        y ← Root(T)
        while y ≠ null
                prev ← y
                if x < y
                        y ← Left(y)
                else
                        y ← Right(y)
        Parent(x) ← prev
        if x < prev
                Left(prev) ← x
        else
                Right(prev) ← x
```

What happens if it is a duplicate?

Inserting duplicate

$$left(i) < i \le right(i)$$

Insert(T, 14)



Running time

```
BSTInsert(T, x)
if Root(T) = null
        Root(T) ← x
else
        y ← Root(T)
        while y ≠ null
                prev ← y
                if x < y
                        y ← Left(y)
                else
                        y ← Right(y)
        Parent(x) ← prev
        if x < prev
                Left(prev) ← x
        else
                Right(prev) ← x
```

O(height of the tree)

Running time

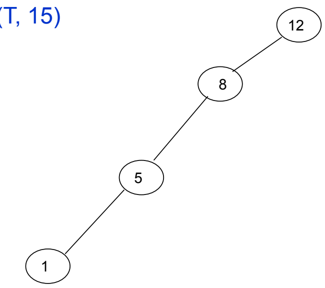
```
BSTInsert(T, x)
if Root(T) = null
        Root(T) ← x
else
        y ← Root(T)
        while y ≠ null
                prev ← y
                if x < y
                        y ← Left(y)
                else
                        y ← Right(y)
        Parent(x) ← prev
        if x < prev
                Left(prev) ← x
        else
                Right(prev) ← x
```

O(height of the tree)

Why not Θ(height of the tree)?

Running time

Insert(T, 15)



Height of the tree

- Worst case: "the twig" – When will this happen?

```
BSTInsert(T, x)
if Root(T) = null
        Root(T) ← x
else
        y ← Root(T)
        while y ≠ null
                prev ← y
                if x < y
                        y ← Left(y)
                else
                        y ← Right(y)
        Parent(x) ← prev
        if x < prev
                Left(prev) ← x
        else
                Right(prev) ← x
```

Height of the tree

- Best case: "complete" – When will this happen?

```
BSTInsert(T, x)
if Root(T) = null
        Root(T) ← x
else
        y ← Root(T)
        while y ≠ null
                prev ← y
                if x < y
                        y ← Left(y)
                else
                        y ← Right(y)
        Parent(x) ← prev
        if x < prev
                Left(prev) ← x
        else
                Right(prev) ← x
```

Height of the tree

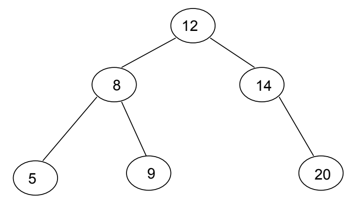
- Average case for random data?

```
BSTInsert(T, x)
if Root(T) = null
      Root(T) ← x
else
      y ← Root(T)
      while y ≠ null
            prev ← y
            if x < y
                  y ← Left(y)
            else
                  y ← Right(y)
      Parent(x) ← prev
      if x < prev
            Left(prev) ← x
      else
            Right(prev) ← x
```

Randomly inserted data into a BST generates a tree on average that is O(log n)

Visiting all nodes

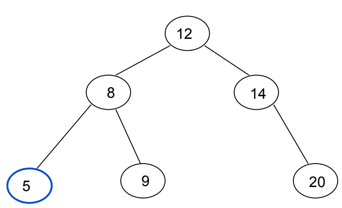
- In sorted order



Visiting all nodes

- In sorted order

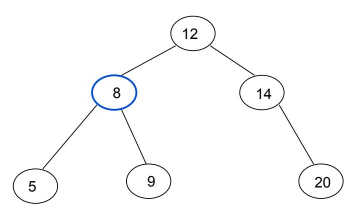
5



Visiting all nodes

- In sorted order

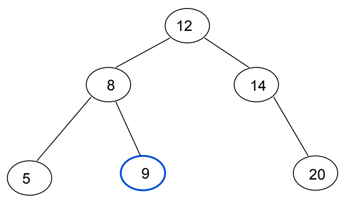
5, 8



Visiting all nodes

- In sorted order

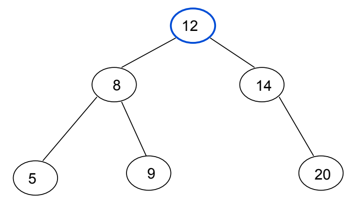
5, 8, 9



Visiting all nodes

- In sorted order

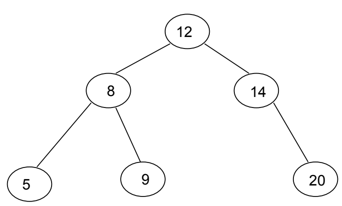
5, 8, 9, 12



Visiting all nodes

- What’s happening?

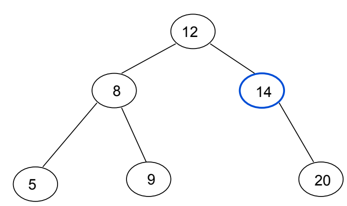
5, 8, 9, 12



Visiting all nodes

- In sorted order

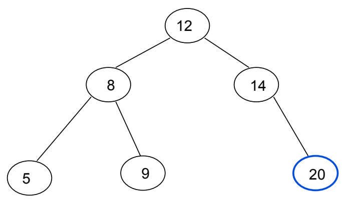
5, 8, 9, 12, 14



Visiting all nodes

- In sorted order

5, 8, 9, 12, 14, 20



Visiting all nodes in order

```
InorderTreeWalk(x)
if x ≠ null
    InorderTreeWalk(Left(x))
    print x
    InorderTreeWalk(Right(x))
```

Visiting all nodes in order

```
InorderTreeWalk(x)
if x ≠ null
    InorderTreeWalk(Left(x))
    print x
    InorderTreeWalk(Right(x))
```

any operation

Is it correct?

```
InorderTreeWalk(x)
if x ≠ null
    InorderTreeWalk(Left(x))
    print x
    InorderTreeWalk(Right(x))
```

- Does it print out all of the nodes in sorted order?

$$left(i) < i \leq right(i)$$

Running time?

```
INORDERTREEWALK(x)
if x ≠ null
    INORDERTREEWALK(LEFT(x))
    print x
    INORDERTREEWALK(RIGHT(x))
```

- Recurrence relation:
  - *j* nodes in the left subtree
  - $n - j - 1$ in the right subtree

$$T(n) = T(j) + T(n - j - 1) + \Theta(1)$$

- Or
  - How much work is done for each call?
  - How many calls?
  - $\Theta(n)$

What about?

```
TREEWALK(x)
if x ≠ null
    print x
    TREEWALK(LEFT(x))
    TREEWALK(RIGHT(x))
```

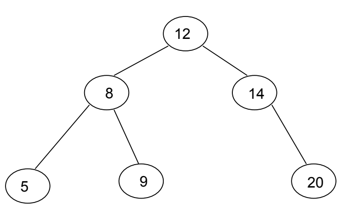
Preorder traversal

```
TREEWALK(x)
if x ≠ null
    print x
    TREEWALK(LEFT(x))
    TREEWALK(RIGHT(x))
```

12, 8, 5, 9, 14, 20

How is this useful?

- Tree copying: insert in to new tree in preorder
- prefix notation: (2+3)*4 -> * + 2 3 4



What about?

```
TREEWALK(x)
if x ≠ null
    TREEWALK(LEFT(x))
    TREEWALK(RIGHT(x))
    print x
```

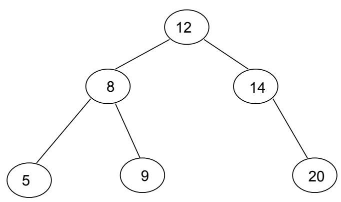
Postorder traversal

```
TREEWALK(x)
  if x ≠ null
      TREEWALK(LEFT(x))
      TREEWALK(RIGHT(x))
      print x
```

5, 9, 8, 20, 14, 12

How is this useful?

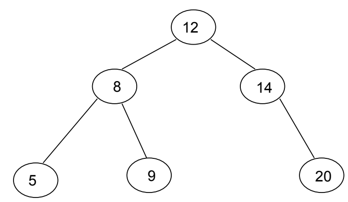
- postfix notation: (2+3)*4 -> 4 3 2 + *
- ?



Min/Max

```
BSTMIN(x)
  if LEFT(x) = null
      return x
  else
      return BSTMIN(LEFT(x))
```

```
ITERATIVEBSTMIN(x)
  while LEFT(x) ≠ null
      x ← LEFT(x)
  return x
```



Running time of min/max?

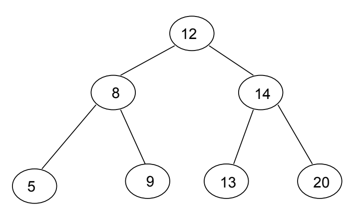
```
BSTMIN(x)
  if LEFT(x) = null
      return x
  else
      return BSTMIN(LEFT(x))
```

```
ITERATIVEBSTMIN(x)
  while LEFT(x) ≠ null
      x ← LEFT(x)
  return x
```

O(height of the tree)

Successor and predecessor

Predecessor(12)? 9

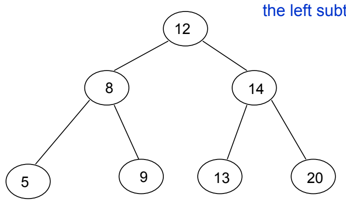


Successor and predecessor

Predecessor in general?

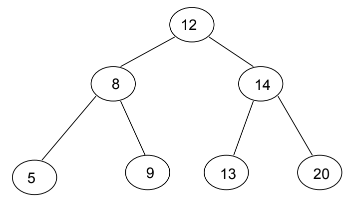
largest node of all those smaller than this node

rightmost element of the left subtree



Successor

Successor(12)? 13

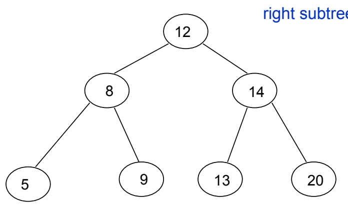


Successor

Successor in general?

smallest node of all those larger than this node

leftmost element of the right subtree

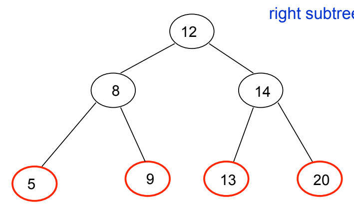


Successor

What if the node doesn't have a right subtree?

smallest node of all those larger than this node

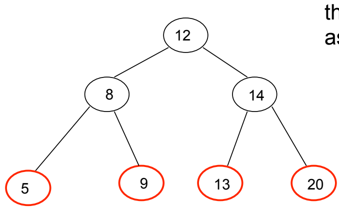
leftmost element of the right subtree



Successor

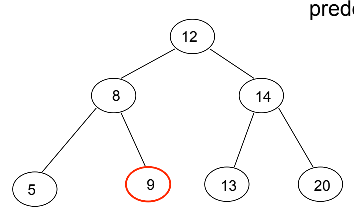
What if the node doesn't have a right subtree?

- node is the largest
- the successor is the node that has x as a predecessor



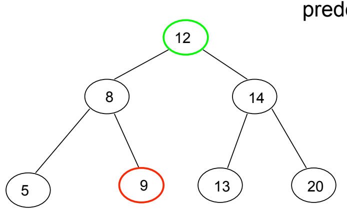
Successor

successor is the node that has x as a predecessor



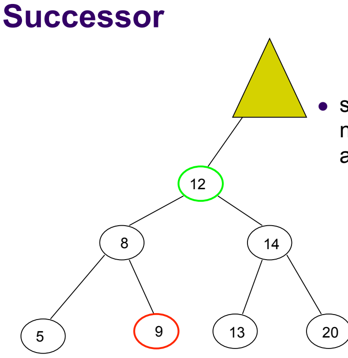
Successor

successor is the node that has x as a predecessor



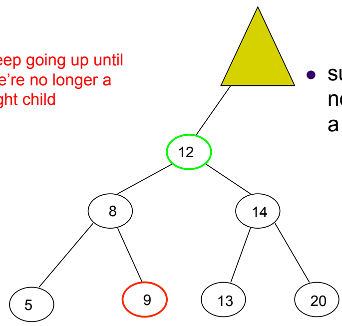
Successor

- successor is the node that has x as a predecessor



Successor

keep going up until we're no longer a right child



- successor is the node that has x as a predecessor

Successor

```
SUCCESSOR(x)
if RIGHT(x) ≠ null
        return BSTMIN(RIGHT(x))
else
        y ← PARENT(x)
        while y ≠ null and x = RIGHT(y)
                x ← y
                y ← PARENT(y)
return y
```

Successor

```
SUCCESSOR(x)
if RIGHT(x) ≠ null
        return BSTMIN(RIGHT(x))
else
        y ← PARENT(x)
        while y ≠ null and x = RIGHT(y)
                x ← y
                y ← PARENT(y)
return y
```

if we have a right subtree, return the smallest of the right subtree

Successor

```
SUCCESSOR(x)
if RIGHT(x) ≠ null
        return BSTMIN(RIGHT(x))
else
        y ← PARENT(x)
        while y ≠ null and x = RIGHT(y)
                x ← y
                y ← PARENT(y)
return y
```

find the node that x is the predecessor of

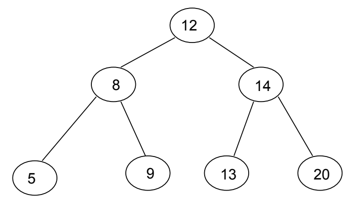
keep going up until we're no longer a right child

Successor running time

O(height of the tree)

```
SUCCESSOR(x)
if RIGHT(x) ≠ null
        return BSTMIN(RIGHT(x))
else
        y ← PARENT(x)
        while y ≠ null and x = RIGHT(y)
                x ← y
                y ← PARENT(y)
return y
```

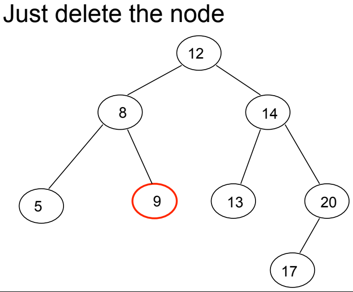
Deletion



Three cases!

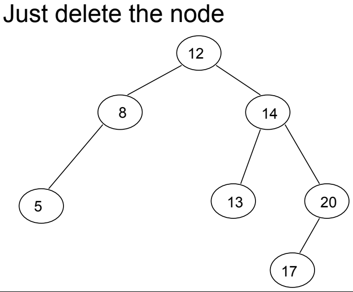
Deletion: case 1

- No children
- Just delete the node



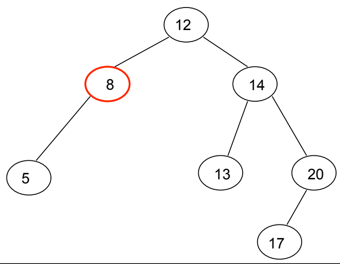
Deletion: case 1

- No children
- Just delete the node



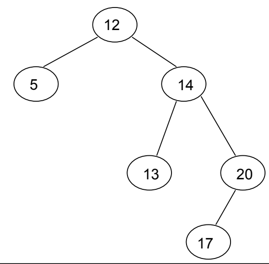
Deletion: case 2

- One child
- Splice out the node



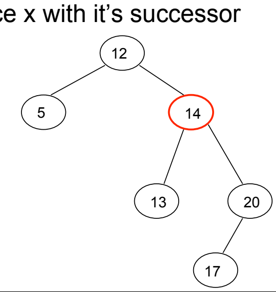
Deletion: case 2

- One child
- Splice out the node



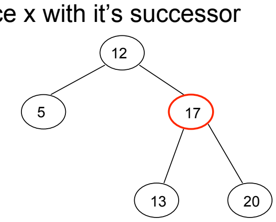
Deletion: case 3

- Two children
- Replace x with it's successor



Deletion: case 3

- Two children
- Replace x with it's successor



Deletion: case 3

- Two children
- Will we always have a successor?
- Why successor?
  - Case 1 or case 2 deletion
  - Larger than the left subtree
  - Less than or equal to right subtree

Height of the tree

- Most of the operations take time O(height of the tree)
- We said trees built from random data have height O(log n), which is asymptotically tight
- Two problems:
  - We can't always insure random data
  - What happens when we delete nodes and insert others after building a tree?

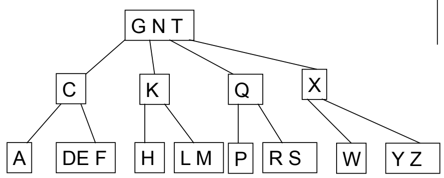
Balanced trees

- Make sure that the trees remain balanced!
  - Red-black trees
  - AVL trees
  - 2-3-4 trees
  - …
- B-trees

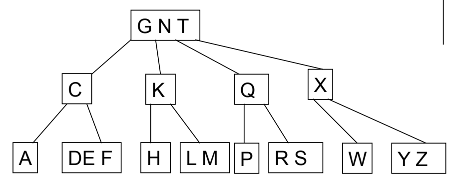
B-tree

- Defined by one parameter: *t*
- Balanced n-ary tree
- Each node contains between *t*-1 and 2*t*-1 keys/data values (i.e. multiple data values per tree node)
  - keys/data are stored in **sorted order**
  - one exception: root can have < t-1 keys
- Each internal node contains between *t* and 2*t* children
  - the keys of a parent **delimit** the values of the children keys
  - For example, if key_i = 15 and key_{i+1} = 25 then child *i* + 1 must have keys between 15 and 25
- all leaves have the same depth

Example B-tree: t = 2

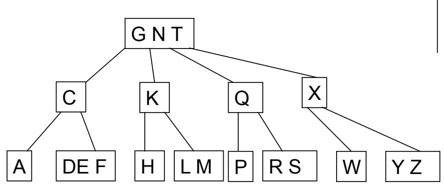


Example B-tree: t = 2



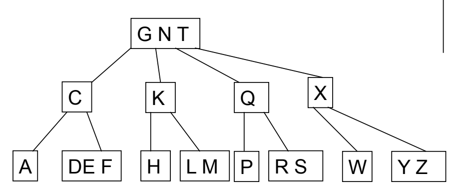
Balanced: all leaves
have the same depth

Example B-tree: t = 2



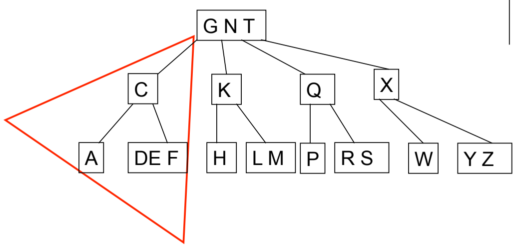
Each node contains between $t-1$ and $2t-1$
keys stored in increasing order

Example B-tree: t = 2



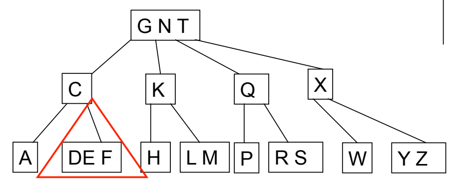
Each node contains between t and $2t$ children

Example B-tree: t = 2



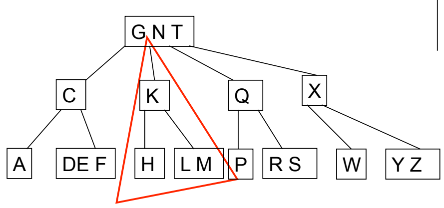
The keys of a parent delimit the values that a child's keys can take

Example B-tree: t = 2



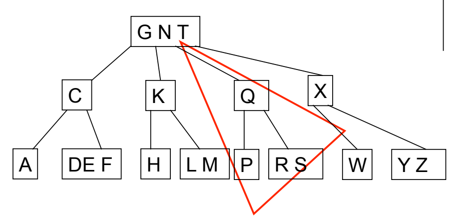
The keys of a parent delimit the values that a child's keys can take

Example B-tree: t = 2



The keys of a parent delimit the values that a child's keys can take

Example B-tree: t = 2



The keys of a parent delimit the values that a child's keys can take

When do we use B-trees over other balanced trees?

- B-trees are generally an **on-disk** data structure
- Memory is limited or there is a large amount of data to be stored
- In the extreme, only one node is kept in memory and the rest on disk
- Size of the nodes is often determined by a page size on disk. Why?
- Databases frequently use B-trees

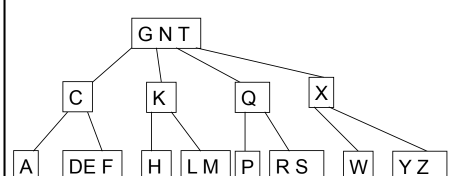
Notes about B-trees

- Because t is generally large, the height of a B-tree is usually small
  - t = 1001 with height 2 can have over one billion values
- We will count both run-time as well as the number of disk accesses. Why?

Height of a B-tree

- B-trees have a similar feeling to BSTs
- We saw for BSTs that most of the operations depended on the height of the tree
- How can we bound the height of the tree?
- We know that nodes must have a minimum number of keys/data items
- For a tree of height h, what is the smallest number of keys?

Minimum number of nodes at each depth?



1 root ☺

2 children

$2t$ children

$2t^{h-1}$ children

In general?

Minimum number of keys/values

root — min. keys per node — min. number of nodes

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1}$$

Minimum number of nodes

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1}$$

$$= 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right)$$

$$= 2t^h - 1$$

so,

$$t^h \leq (n+1)/2$$

$$h \leq \log_t \frac{(n+1)}{2}$$

Searching B-Trees

Find value *k* into B-Tree node *x*

number of keys — key[i] — child[i]

```
B-TreeSearch(x, k)
i ← 1
while i ≤ n(x) and k > K_x[i]
        i ← i + 1
if i ≤ n(x) and k = K_x[i]
        return (x, i)
if Leaf(x)
        return null
else
        DiskRead(C_x[i])
        return B-TreeSearch(C_x[i], k)
```

Searching B-Trees

```
B-TreeSearch(x, k)
i ← 1
while i ≤ n(x) and k > K_x[i]
        i ← i + 1
if i ≤ n(x) and k = K_x[i]
        return (x, i)
if Leaf(x)
        return null
else
        DiskRead(C_x[i])
        return B-TreeSearch(C_x[i], k)
```

make disk reads explicit

Searching B-Trees

B-TreeSearch(x, k)

$i \leftarrow 1$
while $i \leq n(x)$ and $k > K_x[i]$
    $i \leftarrow i + 1$
if $i \leq n(x)$ and $k = K_x[i]$
    return (x, i)
if Leaf(x)
    return *null*
else
    DiskRead$(C_x[i])$
    return B-TreeSearch$(C_x[i], k)$

iterate through the sorted keys and find the correct location

Searching B-Trees

B-TreeSearch(x, k)

$i \leftarrow 1$
while $i \leq n(x)$ and $k > K_x[i]$
    $i \leftarrow i + 1$
if $i \leq n(x)$ and $k = K_x[i]$
    return (x, i)
if Leaf(x)
    return *null*
else
    DiskRead$(C_x[i])$
    return B-TreeSearch$(C_x[i], k)$

if we find the value in this node, return it

Searching B-Trees

B-TreeSearch(x, k)

$i \leftarrow 1$
while $i \leq n(x)$ and $k > K_x[i]$
    $i \leftarrow i + 1$
if $i \leq n(x)$ and $k = K_x[i]$
    return (x, i)
if Leaf(x)
    return *null*
else
    DiskRead$(C_x[i])$
    return B-TreeSearch$(C_x[i], k)$

if it' s a leaf and we didn' t find it, it' s not in the tree

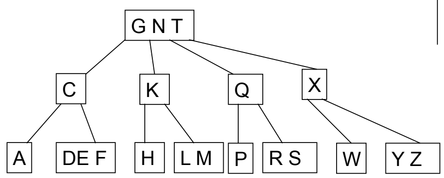
Searching B-Trees

B-TreeSearch(x, k)

$i \leftarrow 1$
while $i \leq n(x)$ and $k > K_x[i]$
    $i \leftarrow i + 1$
if $i \leq n(x)$ and $k = K_x[i]$
    return (x, i)
if Leaf(x)
    return *null*
else
    DiskRead$(C_x[i])$
    return B-TreeSearch$(C_x[i], k)$

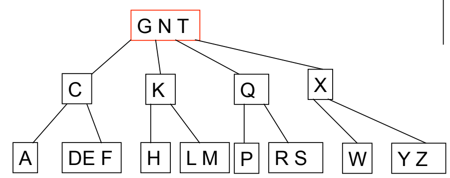
Recurse on the proper child where the value is between the keys

Search example: R



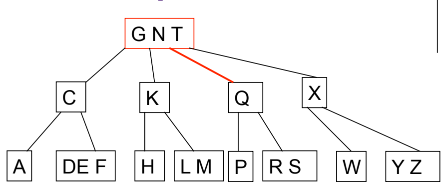

```
B-TreeSearch(x, k)
i ← 1
while i ≤ n(x) and k > K_x[i]
        i ← i + 1
if i ≤ n(x) and k = K_x[i]
        return (x, i)
if Leaf(x)
        return null
else
        DiskRead(C_x[i])
        return B-TreeSearch(C_x[i], k)
```

Search example: R




```
B-TreeSearch(x, k)
i ← 1
while i ≤ n(x) and k > K_x[i]
        i ← i + 1
if i ≤ n(x) and k = K_x[i]
        return (x, i)
if Leaf(x)
        return null
else
        DiskRead(C_x[i])
        return B-TreeSearch(C_x[i], k)
```

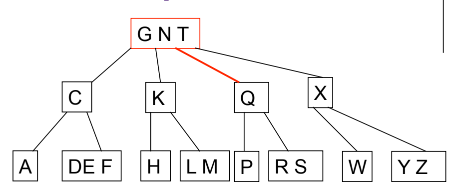
Search example: R




```
B-TreeSearch(x, k)
i ← 1
while i ≤ n(x) and k > K_x[i]
        i ← i + 1
if i ≤ n(x) and k = K_x[i]
        return (x, i)
if Leaf(x)
        return null
else
        DiskRead(C_x[i])
        return B-TreeSearch(C_x[i], k)
```

find the correct location

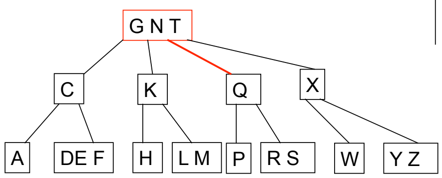
Search example: R




```
B-TreeSearch(x, k)
i ← 1
while i ≤ n(x) and k > K_x[i]
        i ← i + 1
if i ≤ n(x) and k = K_x[i]
        return (x, i)
if Leaf(x)
        return null
else
        DiskRead(C_x[i])
        return B-TreeSearch(C_x[i], k)
```

the value is not in this node

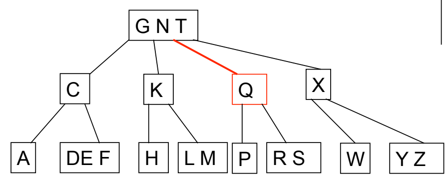
Search example: R




```
B-TreeSearch(x, k)
i ← 1
while i ≤ n(x) and k > K_x[i]
        i ← i + 1
if i ≤ n(x) and k = K_x[i]
        return (x, i)
if Leaf(x)
        return null
else
        DiskRead(C_x[i])
        return B-TreeSearch(C_x[i], k)
```

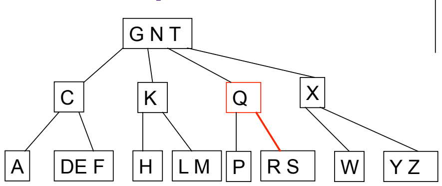
this is not a leaf node

Search example: R




```
B-TreeSearch(x, k)
i ← 1
while i ≤ n(x) and k > K_x[i]
        i ← i + 1
if i ≤ n(x) and k = K_x[i]
        return (x, i)
if Leaf(x)
        return null
else
        DiskRead(C_x[i])
        return B-TreeSearch(C_x[i], k)
```

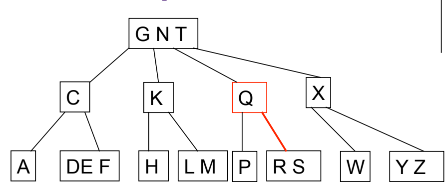

Search example: R




```
B-TreeSearch(x, k)
i ← 1
while i ≤ n(x) and k > K_x[i]
        i ← i + 1
if i ≤ n(x) and k = K_x[i]
        return (x, i)
if Leaf(x)
        return null
else
        DiskRead(C_x[i])
        return B-TreeSearch(C_x[i], k)
```


find the correct location

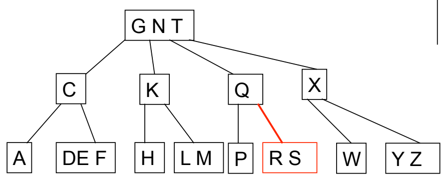
Search example: R




```
B-TreeSearch(x, k)
i ← 1
while i ≤ n(x) and k > K_x[i]
        i ← i + 1
if i ≤ n(x) and k = K_x[i]
        return (x, i)
if Leaf(x)
        return null
else
        DiskRead(C_x[i])
        return B-TreeSearch(C_x[i], k)
```

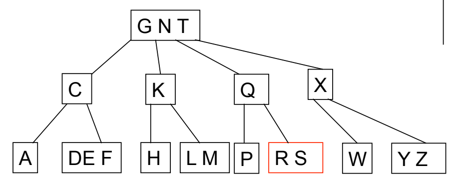

not in this node and this is not a leaf

Search example: R




```
B-TreeSearch(x, k)
i ← 1
while i ≤ n(x) and k > K_x[i]
        i ← i + 1
if i ≤ n(x) and k = K_x[i]
        return (x, i)
if Leaf(x)
        return null
else
        DiskRead(C_x[i])
        return B-TreeSearch(C_x[i], k)
```

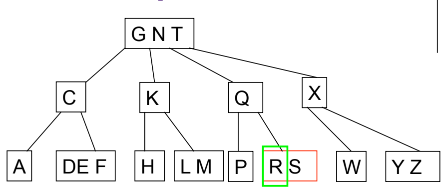
Search example: R




```
B-TreeSearch(x, k)
i ← 1
while i ≤ n(x) and k > K_x[i]
        i ← i + 1
if i ≤ n(x) and k = K_x[i]
        return (x, i)
if Leaf(x)
        return null
else
        DiskRead(C_x[i])
        return B-TreeSearch(C_x[i], k)
```

find the correct location

Search example: R



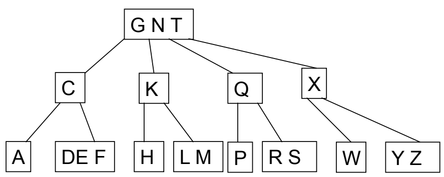

```
B-TreeSearch(x, k)
i ← 1
while i ≤ n(x) and k > K_x[i]
        i ← i + 1
if i ≤ n(x) and k = K_x[i]
        return (x, i)
if Leaf(x)
        return null
else
        DiskRead(C_x[i])
        return B-TreeSearch(C_x[i], k)
```

Search running time

- How many calls to BTreeSearch?
  - O(height of the tree)
  - $O(\log_t n)$
- Disk accesses?
  - One for each call – $O(\log_t n)$
- Computational time?
  - O(t) keys per node
  - linear search
  - $O(t \log_t n)$
- Why not binary search to find key in a node?

```
B-TreeSearch(x, k)
i ← 1
while i ≤ n(x) and k > K_x[i]
        i ← i + 1
if i ≤ n(x) and k = K_x[i]
        return (x, i)
if Leaf(x)
        return null
else
        DiskRead(C_x[i])
        return B-TreeSearch(C_x[i], k)
```

BST-Insert



B-Tree insert

- Starting at root, follow the *search* path down the tree
  - If the node is full (contains 2*t* - 1 keys)
    - split the keys into two nodes around the median value
    - add the median value to the parent node
  - If the node is a leaf, insert it into the correct spot
- Observations
  - Insertions **always** happens in the leaves
  - When does the height of a B-tree grow?
  - Why do we know it's always ok when we're splitting a node to insert the median value into the parent?

Insertion: t = 2

G C N A H E K Q M F W L T Z D P R X Y S

Insertion: t = 2

G C N A H E K Q M F W L T Z D P R X Y S

Insertion: t = 2

G C N A H E K Q M F W L T Z D P R X Y S

C G

Insertion: t = 2

G C N A H E K Q M F W L T Z D P R X Y S

C G N

Insertion: t = 2

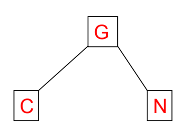
G C N A H E K Q M F W L T Z D P R X Y S

C G N

Node is full, so split

Insertion: t = 2

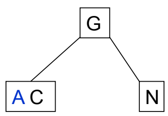
G C N A H E K Q M F W L T Z D P R X Y S



Node is full, so split

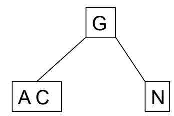
Insertion: t = 2

G C N A H E K Q M F W L T Z D P R X Y S



Insertion: t = 2

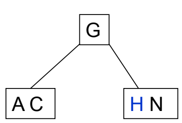
G C N A H E K Q M F W L T Z D P R X Y S



?

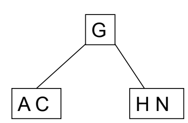
Insertion: t = 2

G C N A H E K Q M F W L T Z D P R X Y S



Insertion: t = 2

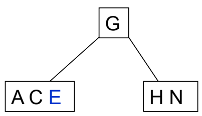
G C N A H E K Q M F W L T Z D P R X Y S



?

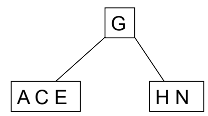
Insertion: t = 2

G C N A H E K Q M F W L T Z D P R X Y S



Insertion: t = 2

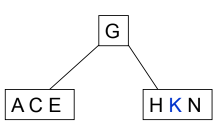
G C N A H E K Q M F W L T Z D P R X Y S



?

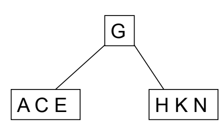
Insertion: t = 2

G C N A H E K Q M F W L T Z D P R X Y S



Insertion: t = 2

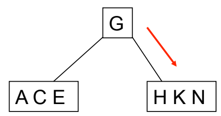
G C N A H E K Q M F W L T Z D P R X Y S



?

Insertion: t = 2

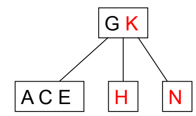
G C N A H E K Q M F W L T Z D P R X Y S



Node is full, so split

Insertion: t = 2

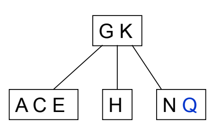
G C N A H E K Q M F W L T Z D P R X Y S



Node is full, so split

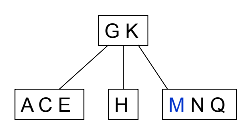
Insertion: t = 2

G C N A H E K Q M F W L T Z D P R X Y S



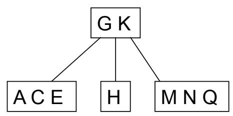
Insertion: t = 2

G C N A H E K Q M F W L T Z D P R X Y S



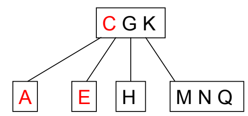
Insertion: t = 2

G C N A H E K Q M F W L T Z D P R X Y S



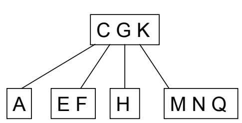
Insertion: t = 2

G C N A H E K Q M F W L T Z D P R X Y S



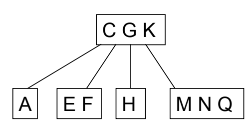
Insertion: t = 2

G C N A H E K Q M F W L T Z D P R X Y S



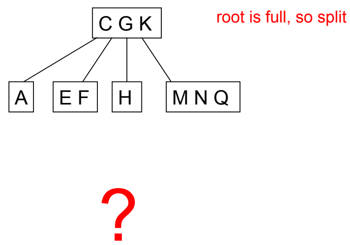
Insertion: t = 2

G C N A H E K Q M F W L T Z D P R X Y S



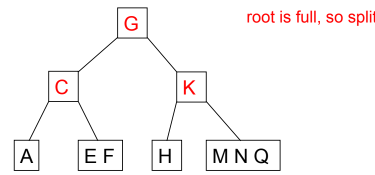
Insertion: t = 2

G C N A H E K Q M F W L T Z D P R X Y S



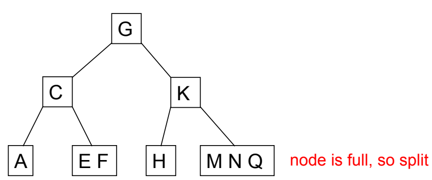
Insertion: t = 2

G C N A H E K Q M F W L T Z D P R X Y S



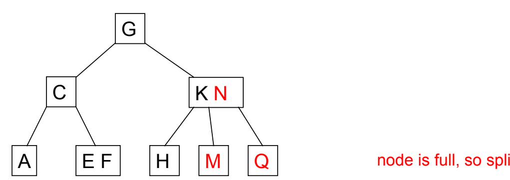
Insertion: t = 2

G C N A H E K Q M F W L T Z D P R X Y S



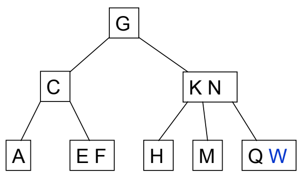
Insertion: t = 2

G C N A H E K Q M F W L T Z D P R X Y S



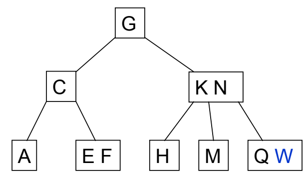
Insertion: t = 2

G C N A H E K Q M F W L T Z D P R X Y S



Insertion: t = 2

G C N A H E K Q M F W …



Correctness of insert

- Starting at root, follow *search* path down the tree
  - If the node is full (contains 2*t* - 1 keys), split the keys around the median value into two nodes and add the median value to the parent node
  - If the node is a leaf, insert it into the correct spot
- Does it add the value in the correct spot?
  - Follows the correct *search* path
  - Inserts in correct position

Correctness of insert

- Starting at root, follow *search* path down the tree
  - If the node is full (contains 2*t* - 1 keys), split the keys around the median value into two nodes and add the median value to the parent node
  - If the node is a leaf, insert it into the correct spot
- Do we maintain a proper B-tree?
  - Maintain t-1 to 2t-1 keys per node?
    - Always split full nodes when we see them
    - Only split full nodes
  - All leaves at the same level?
    - Only add nodes at leaves

Insert running time

- Without any splitting?
  - Similar to BTreeSearch, with one extra disk write at the leaf
  - $O(\log_t n)$ disk accesses
  - $O(t \log_t n)$ computation time

When a node is split

- How many disk accesses?
  - 3 disk write operations
    - 2 for the new nodes created by the split (one is reused, but must be updated)
    - 1 for the parent node to add median value
- Runtime to split a node?
  - $O(t)$ – iterating through the elements a few times since they're already in sorted order
- Maximum number of nodes split for a call to insert?
  - O(height of the tree)

Running time of insert

- $O(\log_t n)$ disk accesses
- $O(t \log_t n)$ computational costs

Deleting a node from a B-tree

- Similar to insertion
  - must make sure we maintain B-tree properties (i.e. all leaves same depth and key/node restrictions)
  - Proactively move a key from a child to a parent if the parent has t-1 keys
- $O(\log_t n)$ disk accesses
- $O(t \log_t n)$ computational costs

Summary of operations

- Search, Insertion, Deletion
  - disk accesses: $O(\log_t n)$
  - computation: $O(t \log_t n)$
- Max, Min
  - disk accesses: $O(\log_t n)$
  - computation: $O(\log_t n)$
- Tree traversal
  - disk accesses: if 2t ~ page size: O(minimum # pages to store data)
  - Computation: $O(n)$