Lecture 9: March 8, 2001

Today:

- Hash deletion in $O(1)$ time
- Binary Search Trees
- BST Search
- BST Insert
- BST Delete

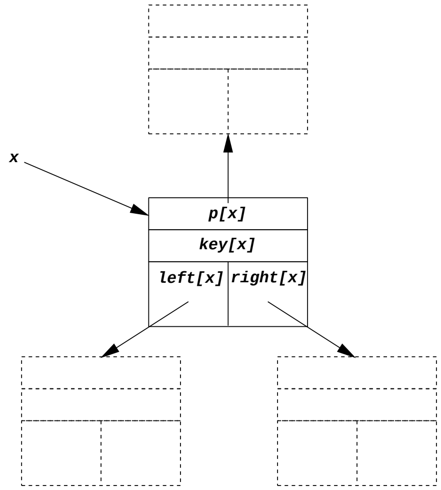
Hash deletion in $O(1)$ time

```
node *Insert( T, char *s ) {
  int h = hash(s);
  node *ph = malloc( sizeof ( char * ) + 2 * sizeof ( void * ));
  ph.s = s;
  ph.prev = NIL;
  ph.next = T[h].head;
  if ( T(h).head ) T[h].head.prev = ph;
  T[h].head = ph;
  return ph;
}

node *Search ( T, char *s ) {
  int h = hash(x);
  node *ph = T[h].head;
  while ( ph )
    if ( !strcmp ( s, ph.s ) ) return ph;
    else ph = ph.next;
  return ph;
}

void SearchAndDelete ( T, char *s ) {
  node *ph = Search ( T, s );
  if ( ph ) {
    ph.next.prev = ph.prev;
    ph.prev.next = ph.next;
    }
}

void Delete ( T, node *ph ) {
  ph.next.prev = ph.prev;
  ph.prev.next = ph.next;
}
```

Binary Search Trees

- Each element x in binary search tree contains:
  - *key*$[x]$ - key stored at x.
  - *left*$[x]$ - pointer to left child of x.
  - *right*$[x]$ - pointer to right child of x.
  - *p[x]* - pointer to parent of x.

BST Element

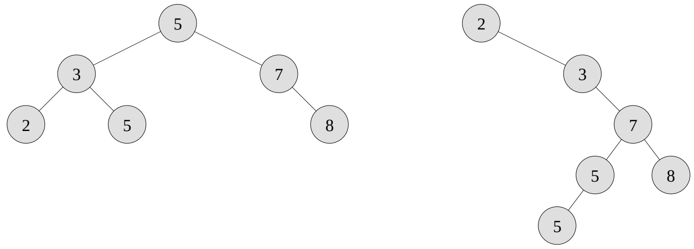


Binary-Search-Tree Property

Keys in binary search tree satisfy the **binary-search-tree property**:

- Let x be a node in a binary search tree.
- y in left subtree of $x \Rightarrow key[y] \leq key[x]$
- y in right subtree of $x \Rightarrow key[x] \leq key[y]$

BST Examples



Inorder Tree Walk

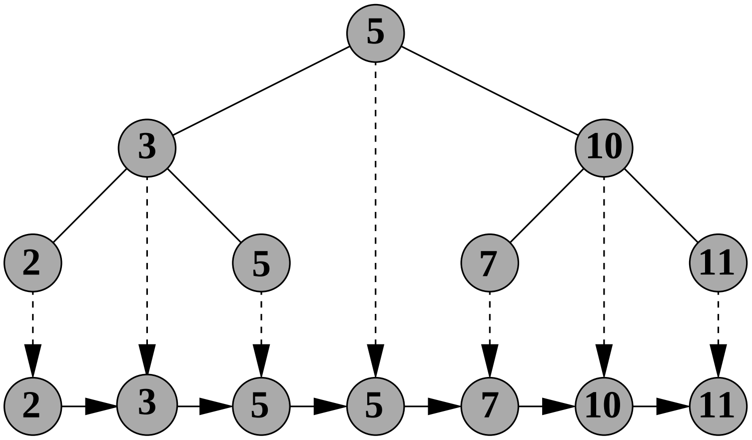
- Can print keys in BST with **inorder tree walk**.
- Key of each node printed between the keys in left subtree and those in right subtree.

```
INORDER-TREE-WALK($x$)
if $x \neq$ NIL
   then INORDER-TREE-WALK($left[x]$)
        print $key[x]$
        INORDER-TREE-WALK($right[x]$)
```

- Prints elements in monotonically increasing order.
- Running Time: $\Theta(n)$

Inorder Tree Walk

- Inorder tree walk can be thought of as a projection of BST onto a line.
- 2^d leaves at depth d occur at x values $\frac{1}{2^{d+1}}, \frac{3}{2^{d+1}}, \ldots, \frac{2^{d+1}-1}{2^{d+1}}$



Other Tree Walks

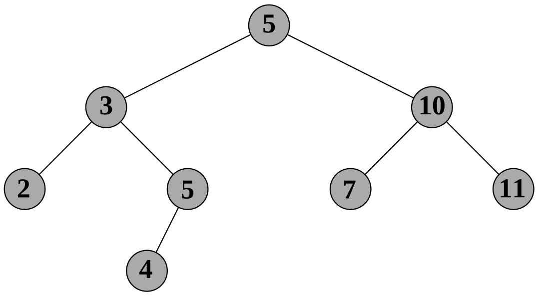
- A **preorder tree walk** processes each node *before* processing its children.

```
PREORDER-TREE-WALK($x$)
if $x \neq$ NIL
    then print $key[x]$
         PREORDER-TREE-WALK($left[x]$)
         PREORDER-TREE-WALK($right[x]$)
```

- A **postorder tree walk** processes each node **after** processing its children.

```
POSTORDER-TREE-WALK($x$)
if $x \neq$ NIL
    then POSTORDER-TREE-WALK($left[x]$)
         POSTORDER-TREE-WALK($right[x]$)
         print $key[x]$
```

Tree Walk Example



- **Inorder walk:**

```
2 3 4 5 5 7 10 11
```

- **Preorder walk:**

```
5 3 2 5 4 10 7 11
```

- **Postorder walk:**

```
2 4 5 3 7 11 10 5
```

Searching in a BST

- To find element with key k in tree T:
- Compare k with $key[root[T]]$
- If $k <$ key[root[T]] search for k in left subtree
- Otherwise, search for k in right subtree

Search Code

Recursive:

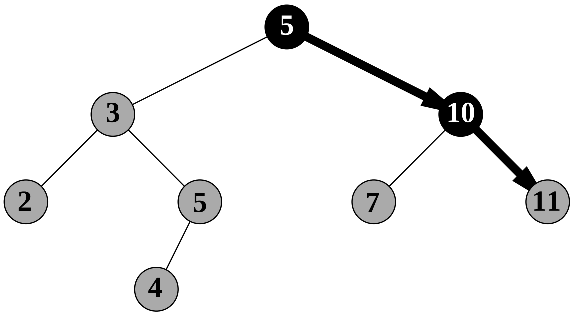
```
SEARCH($T, k$)
$x \leftarrow root[T]$
if $x =$ NIL return NIL
if $k = key[x]$ return $x$
if $k < key[x]$
    then return SEARCH($T, left[x]$)
    else return SEARCH($T, right[x]$)
```

Iterative:

```
SEARCH($T, k$)
$x \leftarrow root[T]$
while $x \neq$ NIL and $k \neq key[x]$
    do if $k < key[x]$
        then $x \leftarrow left[x]$
        else $x \leftarrow right[x]$
return $x$
```

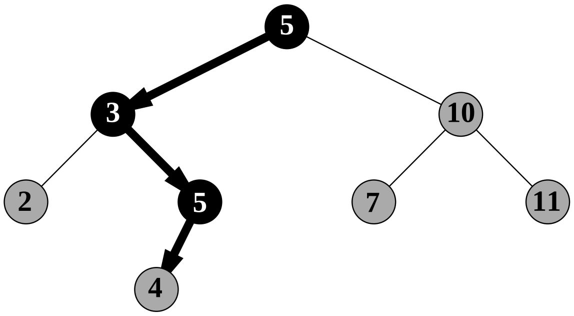
Search Examples

- $\textsc{Search}(T, 11)$



Search Examples

- $\textsc{Search}(T, 4)$



Proof of Correctness

- $S(x) = \{\text{elements in subtree rooted at } x\}$
- **Loop Invariant:**

$$I = (k \in S(\mathit{root}[T]) \Rightarrow k \in S(x))$$

- Initially, I is true.

Proof of Correctness

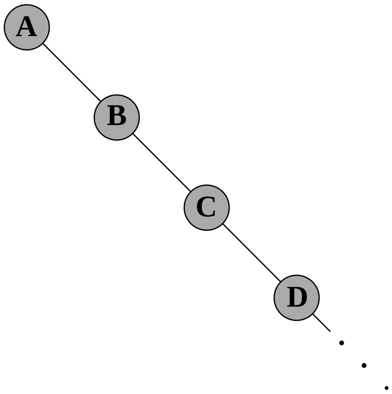
- Inductive Step:
  - $k \in S(\mathit{root}[T]) \Rightarrow k \in S(x)$
  - If $k < \mathit{key}[x]$, then BST property implies $k \notin S(\mathit{right}[x])$.
  - Therefore, $I \Rightarrow k \in S(\mathit{left}[x])$ which implies $k \in S(\mathit{root}[T]) \Rightarrow k \in S(x')$, where $x' = \mathit{left}[x]$.

Proof of Correctness

- Termination occurs since $|S(x)|$ strictly diminishes
- When loop terminates,
  either $x = \text{NIL}$ or $k = key[x]$.
- Therefore, if x in tree, SEARCH will find it.
- Otherwise, SEARCH returns NIL.

Analysis of Search

- Running time is $O(h)$ on tree of height h.
- Worst case running time is $\Theta(n)$.



BST Insertion

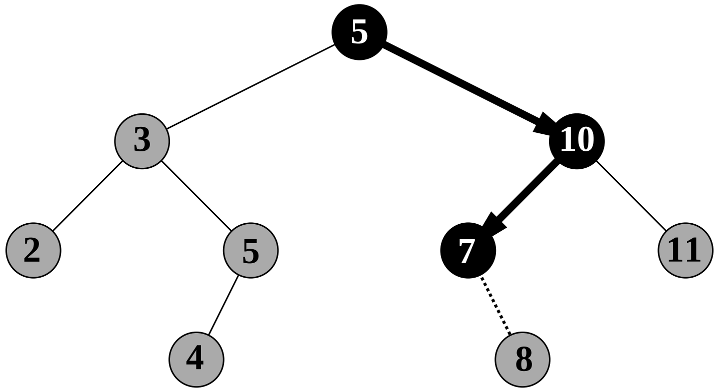
- Similar to SEARCH
- INSERT takes an element z whose right and left children are NIL and inserts it into T.
- Find place in T where z belongs, using code similar to that of SEARCH.
- Add z there.

Insert Code

```
INSERT($T, z$)
  $y \leftarrow$ NIL
  $x \leftarrow root[T]$
  while $x \neq$ NIL
      do $y \leftarrow x$
         if $key[z] < key[x]$
            then $x \leftarrow left[x]$
            else $x \leftarrow right[x]$
  $p[z] \leftarrow y$
  if $y =$ NIL
     then $root[T] \leftarrow z$
     else if $key[z] < key[y]$
            then $left[y] \leftarrow z$
            else $right[y] \leftarrow z$
```

BST Insertion Example

- Insert element with key 8 into tree.



BST Sorting Algorithm

- Can use INSERT and INORDER-TREE-WALK to sort list of n elements, A.

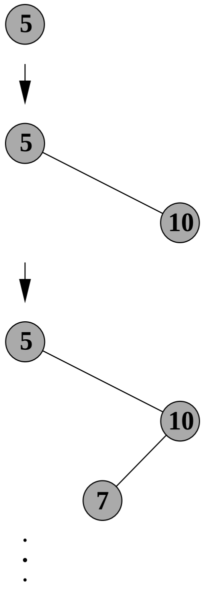
```
BST-SORT
  $root[T] \leftarrow$ NIL
  for $i \leftarrow 1$ to $n$
      do INSERT$(T, A[i])$
  INORDER-TREE-WALK$(root[T])$
```

Sorting Example

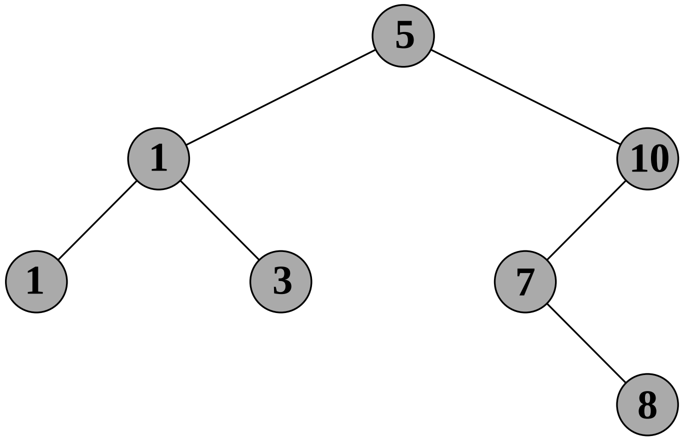
Sort the following numbers:

`5 10 7 1 3 1 8`

- Build binary search tree.



Sorting Example Cont.



- INORDER-TREE-WALK

`1 1 3 5 7 8 10`

Sorting Analysis

- Compare with QUICKSORT partitioning around first (or n^{th} element
- Same comparisons! (Different order).
- Therefore, $O(n \lg n)$ expected time on random input permutation.
- Average tree height is $O(\lg n)$ (recitation)

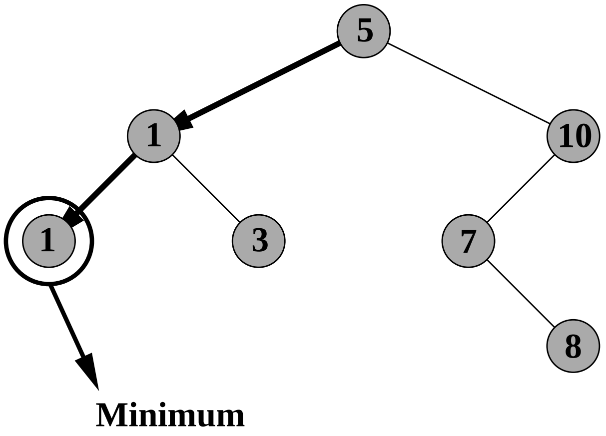
Example: 3 1 8 2 6 7 5

Minimum

- **Problem:** Find minimum key in tree rooted at x.
- **Solution:** Follow left branches.

```
MINIMUM($x$)
  while $left[x] \neq$ NIL
      do $x \leftarrow left[x]$
  return $x$
```

- Running Time = $O(h)$.

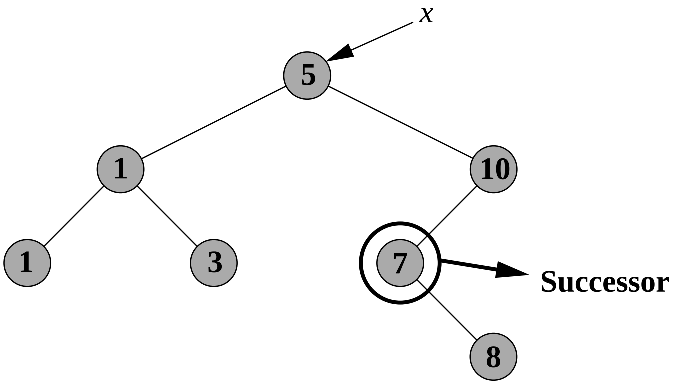


Successor

- **Problem:** Given x, find node with smallest key greater than $key[x]$.
- Two cases.

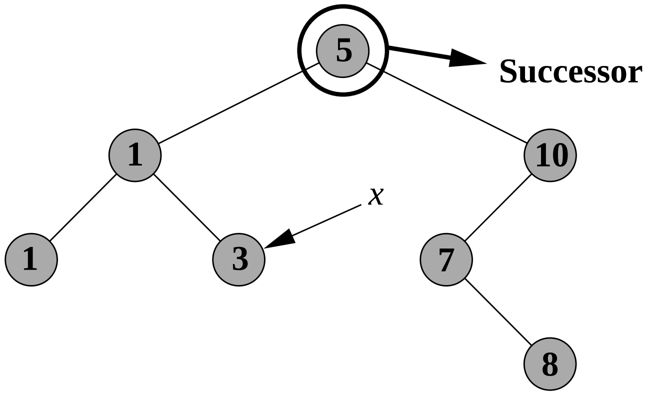
Successor Case 1

- Right subtree of x is nonempty.
- Successor is leftmost node in right subtree.
- (MINIMUM($right[x]$)).



Successor Case 2

- Right subtree of x is empty.
- Successor is *lowest* ancestor of x whose left child is also an ancestor of x.



- Careful! "Successor" is defined as the element next encountered by preorder traversal!

Successor Pseudocode

```
SUCCESSOR($x$)
if $right[x] \neq$ NIL
    then return TREE-MINIMUM($right[x]$)
$y \leftarrow p[x]$
while $y \neq$ NIL and $x = right[y]$
      do $x \leftarrow y$
         $y \leftarrow p[y]$
return $y$
```

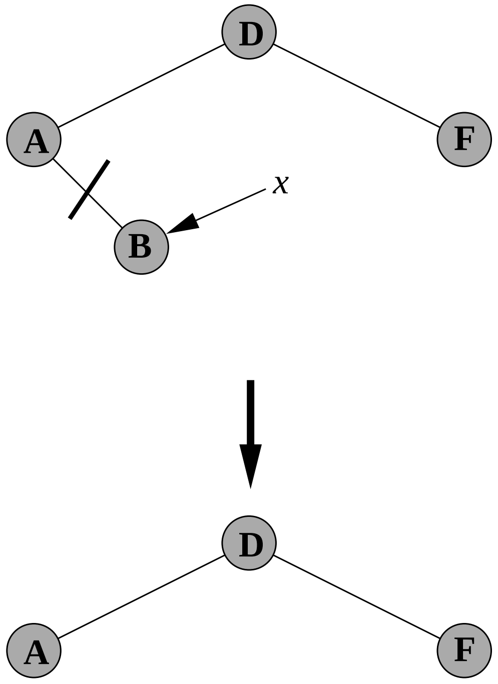
- Time $= O(h)$.

Deletion

- **Problem:** Delete node x from BST.
- 3 cases
  - x has no children.
  - x has one child.
  - x has two children.

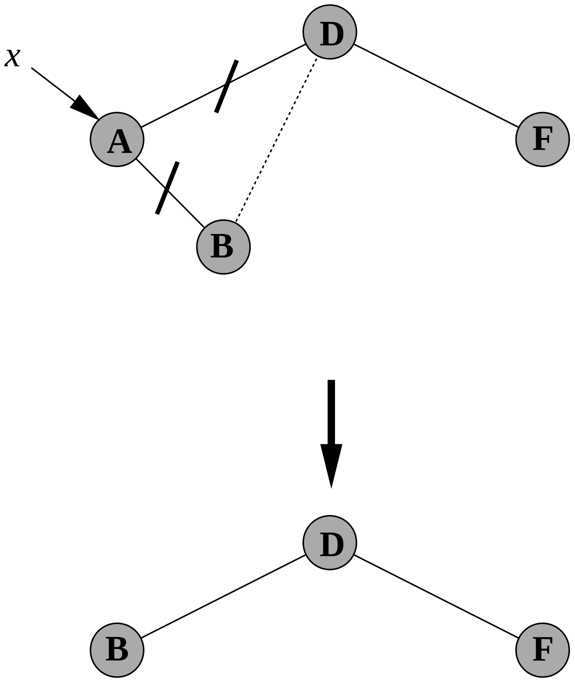
Case 1

- If x has no children, just remove x.



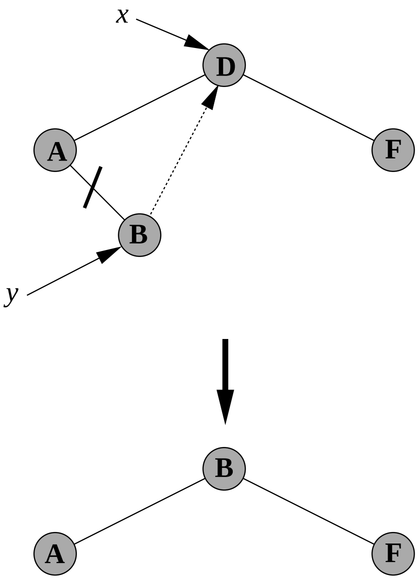
Case 2

- If x has exactly one child, then make $p[x]$ point to that child.



Case 3

- If x has 2 children, then we find its successor (or predecessor) y.
- Remove y. (Note y has at most one child).
- Replace x with y.



Delete Pseudocode

```
DELETE($T, z$)
if $left[z] = $ NIL or $right[z] = $ NIL
    then $y \leftarrow z$
    else $y \leftarrow$ TREE-SUCCESSOR($z$)
if $left[y] \neq$ NIL
    then $x \leftarrow left[y]$
    else $x \leftarrow right[y]$
if $x \neq$ NIL
    then $p[x] \leftarrow p[y]$
if $p[y] = $ NIL
    then $root[T] \leftarrow x$
    else if $y = left[p[y]]$
            then $left[p[y]] \leftarrow x$
            else $right[p[y]] \leftarrow x$
if $y \neq z$
    then $key[z] \leftarrow key[y]$
            ▷ If $y$ has other fields, copy them, too.
return $y$
```

- **Problem:** worst case execution time for dynamic set operations on BST is $\Theta(n)$. No better than linked list!
- **Solution:** "Balanced" search trees– guarantee small height.
- Next time!