The Linked List (LL)

- Each node $x$ in a linked list contains:
  - $key(x)$ - The value stored at $x$.
  - $next(x)$ - Pointer to left child of $x$.
  - The while list is accessed by a head pointer to the first node.
  - Last one points to $NIL$.

Inserting the element into LL

- Find a place for it.
- Point “new’s” next to the next.
- Point “previous’s” next to new.

Removing the element from LL

- Find the element.
- Point “previous’s” next to the next.
- Point previous to new.
Reporting in sorted order

- If you keep the list sorted at all the times the reporting it in sorted order is as simple as iterating through the list and reporting the elements as you encounter them.

Heap

- A heap is a binary tree storing keys, with the following properties:
  - partial order:
    - key (child) < key(parent)
  - left-filled levels:
    - the last level is left-filled
    - the other levels are full

Heap Representations

- left_child(i) = 2i
- right_child(i) = 2i+1
- parent(j) = j div 2
Heap Insertion

- Add the key in the next available spot in the heap.
- Upheap checks if the new node is greater than its parent. If so, it switches the two.
- Upheap continues up the tree.

Upheap terminates when new key is less than the key of its parent or the top of the heap is reached.

(\text{total #switches}) \ll (\text{height of tree} - 1) \approx \log n
Heapify Algorithm

- Assumes L and R sub-trees of i are already Heaps and makes i's sub-tree a Heap:
  - Heapify(A,i,n)
    - If (2i<=n) & (A[2i]>A[i]) Then
      - largest=2i
    - Else largest=i
    - If (2i+1<=n) & (A[2i+1]>A[largest]) Then
      - largest=2i+1
    - If (largest != i) Then
      - Exchange (A[i],A[largest])
    - Heapify(A,largest,n)
  - Endif
- End Heapify

Extracting the Maximum from a Heap:

- Here is the algorithm:
  - Heap-Extract-Max(A)
    - Remove A[1]
    - n=n-1
    - Heapify(a,1,n)
  - End Heap-Extract-Max

Building a Heap

- Builds a heap from an unsorted array:
  - Build_Heap(A,n)
    - For i=floor(n/2) down to 1 do
      - Heapify(A,i,n)
    - End Build_Heap
  - Example:
    - A
      - [4 1 3 2 16 9 10 14 8 7]
Building a Heap (cont’d.)

Finding an Element

- Must look at each element of the array:
  - For i=1 to n do
    - if $A(i) == b$
      - break;

Example:

$A = [X T O G S M N A E R B I]$

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>T</th>
<th>O</th>
<th>G</th>
<th>S</th>
<th>M</th>
<th>N</th>
<th>A</th>
<th>E</th>
<th>R</th>
<th>B</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
Removing an Element

- Must find it first
- After that similar to remove Max

Remove A[i]
n = n - 1
Heapify(a, i, n)

The Binary Search Tree (BST)
- Each node \( x \) in a binary search tree (BST) contains:
  - \( \text{key}[x] \): The value stored at \( x \).
  - \( \text{left}[x] \): Pointer to left child of \( x \).
  - \( \text{right}[x] \): Pointer to right child of \( x \).
  - \( p[x] \): Pointer to parent of \( x \).
**BST- Property**

- Keys in BST satisfy the following properties:
  - Let \( x \) be a node in a BST:
  - If \( y \) is in the left subtree of \( x \) then:
    \[
    \text{key}[y] \leq \text{key}[x]
    \]
  - If \( y \) is in the right subtree of \( x \) then:
    \[
    \text{key}[y] > \text{key}[x]
    \]

**Example:**

- Two valid BST’s for the keys: 2,3,5,7,8.

**In-Order Tree walk**

- Can print keys in BST with in-order tree walk.
- Key of each node printed between keys in left and those in right subtrees.
- Prints elements in monotonically increasing order.
- Running time?
In-Order Traversal

Inorder-Tree-Walk(x)
1: If x!=NIL then
2: Inorder-Tree-Walk(left[x])
3: Print(key[x])
4: Inorder-Tree-Walk(right[x])

What is the recurrence for T(n)?
What is the running time?

In-Order Traversal

- In-Order traversal can be thought of as a projection of BST nodes on an interval.
- At most $2^d$ nodes at level $d=0,1,2,...$

Other Tree Walks

Preorder-Tree-Walk(x)
1: If x!=NIL then
2: Print(key[x])
3: Preorder-Tree-Walk(left[x])
4: Preorder-Tree-Walk(right[x])

Postorder-Tree-Walk(x)
1: If x!=NIL then
2: Postorder-Tree-Walk(left[x])
3: Postorder-Tree-Walk(right[x])
4: Print(key[x])
Searching in BST:

- To find element with key $k$ in tree $T$:
  - Compare $k$ with the root
  - If $k < \text{key}[	ext{root}(T)]$ search for $k$ in the left sub-tree
  - Otherwise, search for $k$ in the right sub-tree

```plaintext
Search(T, k)
1: x = root(T)
2: If x = NIL then return("not found")
3: If k < key(x) then return("found the key")
4: If k > key(x) then Search(left(x), k)
5: else Search(right(x), k)
```

Examples:

- Search($T, 11$)

```
   3
  2 5
  1 4
```

- Search($T, 6$)

```
   3
  2 5
  1 4
```

BST Insertion

- Basic idea: similar to search.
- BST-Insert:
  - Take an element $z$ (whose right and left children are NIL) and insert it into $T$.
  - Find a place where $z$ belongs, using code similar to that of Search.
  - Add $z$ there.
**Insert Key**

```
BST-Insert(T, z)
1: y = NIL
2: x = root(T)
3: While x != NIL do
4:   y = x
5:     if key[z] < key[x] then
6:         x = left[x]
7:     else
8:         x = right[x]
9:     if y = NIL then
10:        root(T) = z
11:     else
12:        if key[z] < key[y] then
13:            left[y] = z
14:        else
15:            right[y] = z
```

**Locating the Minimum**

```
BST-Minimum(T)
1: x = root(T)
2: While left[x] != NIL do
3:   x = left[x]
4: return x
```

**Application: Sorting**

```
Can use BST-Insert and Inorder-Tree-Walk to sort list of n numbers

BST-Sort
1: root(T) = NIL
2: for i = 1 to n do
3:   BST-Insert(T, A[i])
4: Inorder-Tree-Walk(T)

Sort Input: 5, 10, 3, 5, 7, 5, 4, 8
Inorder Walk: 3, 4, 5, 5, 7, 8, 10
```
Successor

Given \( x \), find node with smallest key greater than \( \text{key}(x) \). Here are two cases depending on right subtree of \( x \).

- **Successor Case 1:** The right subtree of \( x \) is not empty. Successor is leftmost node in right subtree. That is, we must return \( \text{BST-Minimum}(\text{right}(x)) \).

- **Successor Case 2:** The right subtree of \( x \) is empty. Successor is lowest ancestor of \( x \). Observe that, “Successor” is defined as the element encountered by inorder traversal.

Deletion

- Delete a node \( x \) from tree \( T \).
  - Case 1: \( x \) has no children.
Deletion:

Case 2: \( x \) has one child (call it \( y \)). Make \( p[x] \) to replace \( y \) instead of \( x \) as its child, and make \( p[x] \) to be \( p[y] \).

![Diagram of tree with one child replaced]

Deletion:

Case 3: \( x \) has two children:
- Find its successor (or predecessor) \( y \).
- Remove \( y \). (Note \( y \) has at most one child, why?)
- Replace \( x \) by \( y \).

![Diagram of tree with successor replaced]

Delete Procedure

```
BST-Delete(T, z)
1: If (left(z)=NIL) or (right(z)=NIL) then
2: \( y = z \)
3: else \( y = \text{BST-Successor}(z) \)
4: If left(y)=NIL then
5: \( x = \text{left}(y) \)
6: else \( x = \text{right}(y) \)
7: If \( x = \text{left}(y) \) then \( \text{left}(p[y]) = x \)
8: else \( \text{right}(p[y]) = x \)
9: if \( y \neq z \) then \( \text{key}(z) = \text{key}(y) \)
10: return \( y \)
```