HEAPS AND TREES
Converting An Array to a Heap

• To convert an array to a heap:
• Key idea: make heaps of subtrees and combine subtrees with new root node using heapify()
• Base case: All leaf nodes are valid heaps
• Begin combining heaps with first non-leaf node
Converting An Array to a Heap

- All leaf nodes are valid heaps.
- Begin at first non-leaf node and continue to decrease location until the root, calling heapify at each location
  - Start: Heapify(Loc. 4)

Leafs are valid heaps by definition
Converting An Array to a Heap

```
  7
 / \
(9) (14)
 /  /  \
(10)(35)(28)(18)
```

Array:
```
em  7  9  14  10  35  28  18  19
```
Build-Heap Run-Time

• To build a heap from an arbitrary array require \( n \) calls to heapify.
• Heapify takes \( O(\text{height}) \)
• More precisely
  – Since most of the heapify calls are shallow, this can be done in \( O(n) \)
    • \( n/2 \) calls with \( h=1 \)
    • \( n/4 \) calls with \( h=2 \)
    • \( n/8 \) calls with \( h=3 \)
    • Totals: \( 1 \times n/2 + 2 \times n/4 + 3 \times n/8 \)

\[
T(n) = \sum_{h=1}^{\log(n)} h \times n \times \left( \frac{1}{2} \right)^h < n \times \sum_{h=1}^{\infty} \left( \frac{1}{2} \right)^h
\]

• \( T(n) = n \times \theta(c) = \theta(n) \)
Array-based and Link-based

TREE IMPLEMENTATIONS
Array-Based Complete Binary Tree

- Binary tree that is complete (i.e. only the lowest-level contains empty locations and items added left to right) can be stored nicely in an array (let’s say it starts at index 1 and index 0 is empty)
- Can you find the mathematical relation for finding node i's parent, left, and right child?
  - Parent(i) = i/2
  - Left_child(i) = 2*i
  - Right_child(i) = 2*i + 1

Non-complete binary trees require much more bookkeeping to store in arrays...usually link-based approaches are preferred
Link-Based Approaches

- For an arbitrary (non-complete) d-ary tree we need to use pointer-based structures.

```cpp
// Bin. Search Tree  template <typename T>
class BinTree {
  public:
    BinTree();
    ~BinTree();
    void add(const T& v); //
  private:
    shared_ptr<Item<T>> root;
};
```

```cpp
template <typename T>
struct Item {
  T val;
  shared_ptr<Item<T>> left;
  shared_ptr<Item<T>> right;
  shared_ptr<Item<T>> parent;
};
```

Item<T> blueprint:
```plaintext
Item<T> parent
  |    |    |
  | left|    | right
  |     | val |
```
Link-Based Approaches

- Add(5)
- Add(6)
- Add(7)
Recursive Tree Traversals

• A traversal iterates over all nodes of the tree
  – Usually using a depth-first, recursive approach

• Three general traversal orderings
  – Pre-order [Process root then visit subtrees]
  – In-order [Visit left subtree, process root, visit right subtree]
  – Post-order [Visit left subtree, visit right subtree, process root]

```c
// Node definition
struct TNode
{
    int val;
    TNode *left, *right;
};

Preorder(TNode* t)
{
    if t == NULL return
    process(t) // print val.
    Preorder(t->left)
    Preorder(t->right)
}

60 20 10 30 25 50 80

Inorder(TNode* t)
{
    if t == NULL return
    Inorder(t->left)
    process(t) // print val.
    Inorder(t->right)
}

10 20 25 30 50 60 80

Postorder(TNode* t)
{
    if t == NULL return
    Postorder(t->left)
    Postorder(t->right)
    process(t) // print val.
}

10 25 50 30 20 80 60
```