

DYNAMIC SETS

Reading: *CLRS* Section B.5, Chapters 10 & 12; *CLR* Section 5.5, Chapters 11 & 13

Data Structure Terminology

We will be studying collections of objects. In this context, an **object** is a record with a **key** field and **satellite data** fields. Every object is identified by a **pointer**; when we say that a function takes an object as an argument or returns one as a result, it is manipulating a pointer to the object. The null pointer, or **nil**, is a distinguished pointer that stands for the absence of an object; it is often used to represent empty lists or trees. We use the term **leaf** to refer to an empty tree and **fringe node** to refer to a tree node whose children are all leaves.

On the next page are pictures showing objects in the context of some of the data structures we will be studying. In the object representations shown, the object has been “merged” with the nodes of the data structure in the sense that some of the satellite data fields are pointers to other objects in the data structure. It is also possible to have objects that are distinct from the nodes of a data structure containing them; such representations are common in so-called functional programming.

We assume that each data structure is assumed to be represented by an **entry point record** with fields that refer to objects as well as auxiliary information. (Such entry point records are *not* shown in the diagram on the next page.) For example, an array has an **elements** field and a **length** field; linked and doubly-linked lists have a **head** field that points to the first object in the list; and trees have a **root** field that points to the root of the tree.

We will assume that new objects are created as follows:

`new Array(n)`

Creates a new array with n slots indexed from 1 to n .

`new ListNode()`

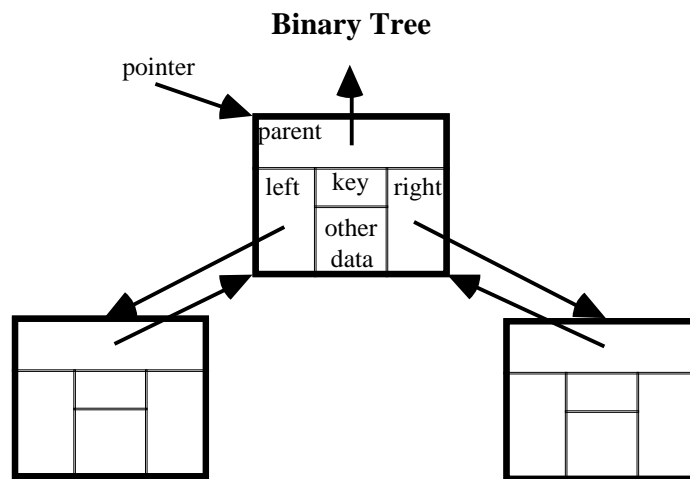
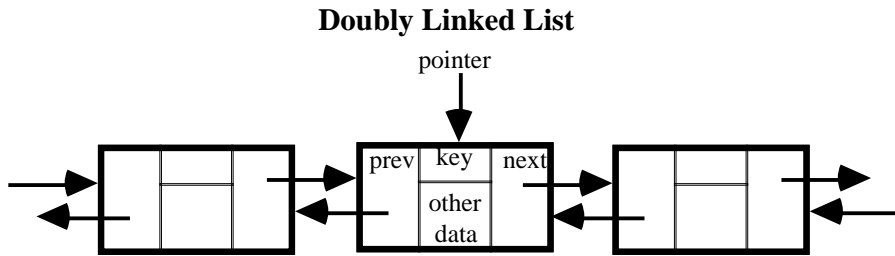
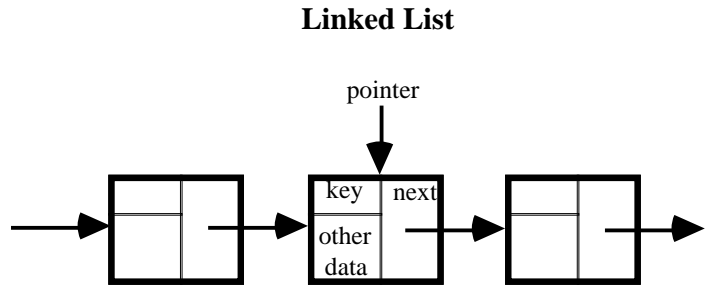
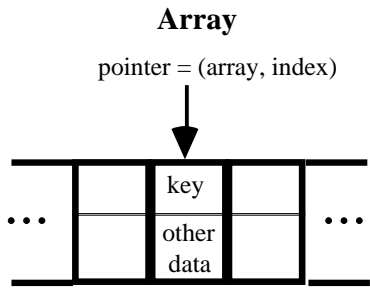
Creates a new list node with fields `key`, `data`, and `next`.

`new DoublyLinkedListNode()`

Creates a new doubly-linked list node with fields `key`, `data`, `next`, and `prev`.

`new BinaryTreeNode()`

Creates a new binary tree node with fields `key`, `data`, `left`, `right`, and `parent`.



Dynamic Sets

A **dynamic set** is an abstract data type for a mutable collection of objects that supports the following seven operations below. Assume that object keys are **distinct** and are related by a **total order**: i.e., any two keys are related by one of $<$, $=$, or $>$. The distinctness restriction is not essential but simplifies the definitions (e.g., without distinctness, we can't refer to *the* maximal object or *the* object with the next largest key).

Below is the contract for the dynamic set data type. Note that `Insert`, `Delete`, `Pred`, and `Succ` take an object pointer as their second argument, not a key. However, it is easy to make versions of these that take keys instead of objects by first using `Search` to find an object with a given key.

`Search(set, key)`

Returns a pointer to an object in `set` with the specified key, or `nil` if there is no such object in `set`.

`Insert(set, obj)`

Destructively updates `set` to add the object `obj`. Insertion may change the fields of `obj` that relate it to other objects in the collection.

`Delete(set, obj)`

Destructively updates `set` to remove the object `obj`.

`Min(set)`

Returns the object in `set` with the smallest key.

`Max(set)`

Returns the object in `set` with the largest key.

`Pred(set, obj)`

Returns the object in `set` whose key directly precedes that of `obj` in the total order of keys in `set`. Returns `nil` if `obj` is the element with the minimal key.

`Succ(set, obj)`

Returns the object in `set` whose key directly follows that of `obj` in the total order of keys in `set`. Returns `nil` if `obj` is the element with the maximal key.

`Eelts(set)`

Returns a linked list of all the objects in `set` ordered by key from low to high.

Linear Implementations of Dynamic Sets

From CS230, you are familiar with representing sets of elements using linear representations like arrays, linked-lists, and doubly-linked lists. These representations are “linear” in the sense that the “nodes” of these data structures are ordered sequentially, as in a line. They are also linear in the sense that several of the operations Search, Insert, Delete, Min, Max, Pred, and Succ take time $\Theta(n)$ for a dynamic set with n elements. (Elt necessary takes time $\Theta(n)$.)

On Problem Set 5, you are asked to determine the best worst-case asymptotic running times for dynamic set operations using sorted and unsorted versions of these linear data structure representations.

Tree Implementations of Dynamic Sets

A better approach for implementing dynamic sets is to arrange the objects in trees. With a little bit of work, it is possible to get $\Theta(\lg(n))$ implementations of all operations Search, Insert, Delete, Min, Max, Pred, and Succ. We will spend several lectures on various ways to achieve these efficient asymptotic running times.

Binary Trees

A **binary tree** is a data structure that is either (1) a leaf (represented as `nil`) or (2) a node with `left` and `right` subtrees (as well as other possible fields).

Typical ways to traverse a binary tree are preorder, inorder, and postorder traversals:

```
Preorder-Traversal(node, function)
  if node = nil then
    function(node)
    Inorder-Traversal(left[node], function)
    Inorder-Traversal(right[node], function)

Inorder-Traversal(node, function)
  if node = nil then
    Inorder-Traversal(left[node], function)
    function(node)
    Inorder-Traversal(right[node], function)

Postorder-Traversal(node, function)
  if node = nil then
    Inorder-Traversal(left[node], function)
    Inorder-Traversal(right[node], function)
    function(node)
```

Traversals can also accumulate the elements of a tree into a list. For example:

```
Inorder-List(node)
  if node = nil then EmptyList
  else Append(Inorder-List(left[node]),
             Prepend(node, InorderList(right[node])))
```

Binary Search Trees

A **binary search tree (BST)** is a binary tree satisfying the **binary search tree property**:

For all objects x in the left subtree of obj , $key[x] < key[obj]$

For all objects y in the right subtree of obj , $key[y] > key[obj]$

There are many binary search trees corresponding to a given set of elements.

E.g., what are the binary search trees for $\{A, B, C\}$?

Tree Sort

Listing the elements of a BST in inorder yields a list of elements sorted by key.

This gives rise to the following sorting algorithm:

```
Tree-Sort(elts)
  Inorder-List(buildBST(elts))
```

Here, *elts* is assumed to be a list of objects, and *buildBST* is assumed to return the BST that results from inserting each element of *elts* into an initially empty binary search tree.

BST Operations I

```
{This algorithm is recursive, but it's easy to make it iterative.}
BST-Search(node, key)
  if (node = nil) or (key = key[node])
    then return node
  if key < key[node]
    then return BST-Search(left[node], key)
    else return BST-Search(right[node], key)

BST-Minimum(node)
  while left[node]  nil
    do node <- left[node]
  return node

{BST-Maximum is symmetric with BST-Minimum}

{The successor is
  (1) The minimum of the right subtree (if it exists)
  (2) The first leftward parent (if it exists)}
BST-Successor(node)
  if right[node]  nil
    then return BST-Minimum(right[node])
  parent-node <- parent[node]
  child-node <- node
  while (parent-node  nil) and (child-node = right[parent-node])
    do child-node <- parent-node
      parent-node <- parent[parent-node]
  {At this point, either
    (1) parent-node is nil
    (2) child-node = left[parent-node]}
  return parent-node

{BST-Predecessor is symmetric with BST-Successor}

BST-Elts(node)
  return Inorder-List(node)
```

BST Operations II

```
{The tree argument is the entry point record for a tree.}
BST-Insert(tree, node)
  previous <- nil
  current <- root[tree]
  while current != nil
    do previous <- current
       if key[node] < key[current]
         then current <- left[current]
         else current <- right[current]
  {current is now nil}
  parent[node] <- previous
  if previous = nil
    then root[tree] <- node
    else if key[node] < key[previous]
      then left[previous] <- node
      else right[previous] <- node

{Different from CLR version.
  Assumes there is a dummy header node at root.}
BST-Delete(tree, node)
  if (left[node] = nil) or (right[node] = nil)
    {Case 1: at least one child of node is nil}
    then if node = left[parent[node]]
      then left[parent[node]] <- Single-Child(node)
      else right[parent[node]] <- Single-Child(node)
    {Case 2: both children of node are non-nil.}
    else succ <- BST-Successor(tree, node)
       {Delete succ from current position.
         Guaranteed to use Case 1.}
       BST-delete(tree, succ)
       {Splice succ into position of node}
       parent[left[node]] <- succ
       left[succ] <- left[node]
       parent[right[node]] <- succ
       right[succ] <- right[node]
       parent[succ] <- parent[node]
       if node = left[parent[node]]
         then left[parent[node]] <- succ
         else right[parent[node]] <- succ

{If one child is nil, returns the other one.
  Returns nil if both children are nil.}
Single-Child(node)
  if left[node] = nil
    then return right[node]
    else return left[node]
```

Running Times of BST Operations

The seven BST operations Search, Insert, Delete, Min, Max, Pred, and Succ have $O(h)$ worst-case running times, where h is the height of the tree.

What is the relationship between the height h and the number of elements n ?

Best case:

Worst case:

Average case:

For the average case, assume that trees are formed by sequentially inserting the elements of a randomly permuted array of n objects into an initially empty BST. Note that there is a close correspondence between the resulting tree and the partitionings of quick sort.

We will spend the next three lectures exploring ways to guarantee that h is $O(\lg(n))$.
