Biostatistics 615/815 Lecture 7:
Elementary Data Structures

Hyun Min Kang

September 27th, 2011
Elementary data structure

Container

A container $T$ is a generic data structure which supports the following three operations for an object $x$.

- $\text{SEARCH}(T, x)$
- $\text{INSERT}(T, x)$
- $\text{DELETE}(T, x)$

Possible types of container

- Arrays
- Linked lists
- Trees
- Hashes
### Average time complexity of container operations

<table>
<thead>
<tr>
<th></th>
<th>Search</th>
<th>Insert</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>SortedArray</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>List</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Tree</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>Hash</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>

- Array or list is simple and fast enough for small-sized data
- Tree is easier to scale up to moderate to large-sized data
- Hash is the most robust for very large datasets
Linked List

- A data structure where the objects are arranged in linear order
- Each object contains the pointer to the next object
- Objects do not exist in consecutive memory space
  - No need to shift elements for insertions and deletions
  - No need to allocate/reallocate the memory space
  - Need to traverse elements one by one
  - Likely inefficient than Array in practice because data is not necessarily localized in memory
- Variants in implementation
  - (Singly-) linked list
  - Doubly-linked list
Example of a linked list

- Example of a doubly-linked list
- Singly-linked list if `prev` field does not exist
Implementation of singly-linked list

```cpp
#include "myListNode.h"

template <class T>
class myList {
    protected:
        myListNode<T>* head; // list only contains the pointer to head
        myList(myList& a) {}; // prevent copying

    public:
        myList() : head(NULL) {} // initially header is NIL
        ~myList();
        void insert(T x); // insert an element x
        int search(T x); // search for an element x and return its location
        bool remove(T x); // delete a particular element
};
```
List implementation: class myListNode

myListNode.h

// myListNode class is only accessible from myList class
template<class T>
class myListNode {
protected:
    T value;       // the value of each element
    myListNode<T>* next;   // pointer to the next element
myListNode(T v, myListNode<T>* n) : value(v), next(n) {} // constructor
~myListNode();
    int search(T x, int curPos);
    myListNode<T>* remove(T x, myListNode<T>*& prevNext);
template <class S> friend class myList; // allow full access to myList class
};
Inserting an element to a list

myList.h

```cpp
template <class T>
void myList<T>::insert(T x) {
    // create a new node, and make them head
    // and assign the original head to head->next
    head = new myListNode<T>(x, head);
}
```

(a) $L$.head

```
prev key next
/ 9 ----- 16 ----- 4 ----- 1 /
```

(b) $L$.head

```
prev key next
/ 25 ----- 9 ----- 16 ----- 4 ----- 1 /
```

(c) $L$.head

```
prev key next
/ 25 ----- 9 ----- 16 ----- 4 ----- 1 /
```
Destructor is required because `new` was used

```cpp
myList.h

```template <class T>
myList<T>::~myList() {
    if ( head != NULL ) {
        delete head;  // delete dependent objects before deleting itself
    }
}
```

```cpp
myListNode.cpp

```template <class T>
myListNode<T>::~myListNode() {
    if ( next != NULL ) {
        delete next;  // recursively calling destructor until the end of the list
    }
}
```
Searching an element from a list

```cpp
myList.h

template <class T>
int myList<T>::search(T x) {
    if ( head == NULL )    return -1;  // NOT_FOUND if empty
    else return head->search(x, 0); // search from the head node
}

myListNode.cpp

template <class T>
// search for element x, and the current index is curPos
int myListNode<T>::search(T x, int curPos) {
    if ( value == x ) return curPos; // if found return current index
    else if ( next == NULL ) return -1;  // NOT_FOUND if reached end-of-list
    else return next->search(x, curPos+1); // recursive call until terminates
}
```
Removing an element from a list

myList.h

```cpp
// myList.h

template <class T>
bool myList<T>::remove(T x) {
  if ( head == NULL )
    return false; // NOT_FOUND if the list is empty
  else {
    // call head->remove will return the object to be removed
    myListNode<T>* p = head->remove(x, head);
    if ( p == NULL ) { // if NOT_FOUND return false
      return false;
    }
    else { // if FOUND, delete the object before returning true
      delete p;
      return true;
    }
  }
}
```
Removing an element from a list

myListNode.h

```cpp
template <class T>
// pass the pointer to [prevElement->next] so that we can change it
myListNode<T>* myListNode<T>::remove(T x, myListNode<T>*& prevNext) {
    if ( value == x ) { // if FOUND
        prevNext = next; // *pPrevNext was this, but change to next
        next = NULL; // disconnect the current object from the list
        return this; // and return it so that it can be destroyed
    }
    else if ( next == NULL ) {
        return NULL; // return NULL if NOT_FOUND
    }
    else {
        return next->remove(x, next); // recursively call on the next element
    }
}
```
Summary - Linked List

- **Class Structure**
  - `myList` class to keep the head node
  - `myListNode` class to store key and pointer to next node
Summary - Linked List

- **Class Structure**
  - `myList` class to keep the head node
  - `myListNode` class to store key and pointer to next node

- **Insert algorithm**: Create a new node as a head node
Summary - Linked List

- **Class Structure**
  - `myList` class to keep the head node
  - `myListNode` class to store key and pointer to next node

- **Insert algorithm**: Create a new node as a head node

- **Search algorithm**
  - Return the index if key matches
  - Otherwise, advance to the next node
Summary - Linked List

- **Class Structure**
  - `myList` class to keep the head node
  - `myListNode` class to store key and pointer to next node

- **Insert algorithm**: Create a new node as a head node

- **Search algorithm**
  - Return the index if key matches
  - Otherwise, advance to the next node

- **Remove algorithm**:
  - Search the element
  - Make the previous node points to the next node
  - Remove the element from the list and destroy it.
Summary - Linked List

- **Class Structure**
  - `myList` class to keep the head node
  - `myListNode` class to store key and pointer to next node

- **Insert algorithm**: Create a new node as a head node

- **Search algorithm**
  - Return the index if key matches
  - Otherwise, advance to the next node

- **Remove algorithm**:
  - Search the element
  - Make the previous node points to the next node
  - Remove the element from the list and destroy it.

- **Q**: What are the advantages and disadvantages between Array and List?
Summary - Linked List

- **Class Structure**
  - `myList` class to keep the head node
  - `myListNode` class to store key and pointer to next node

- **Insert algorithm**: Create a new node as a head node

- **Search algorithm**
  - Return the index if key matches
  - Otherwise, advance to the next node

- **Remove algorithm**:
  - Search the element
  - Make the previous node points to the next node
  - Remove the element from the list and destroy it.

- **Q**: What are the advantages and disadvantages between Array and List?
Binary search tree

Data structure

- The tree contains a root node
- Each node contains
  - Pointers to left and right children
  - Possibly a pointer to its parent
  - And a key value
- Sorted: $\text{left.key} \leq \text{key} \leq \text{right.key}$
- Average $\Theta(\log n)$ complexity for insert, search, remove operations
An example binary search tree
Key algorithms

**INSERT**\( (node, x) \)

1. If the \( node \) is empty, create a leaf node with value \( x \) and return
2. If \( x < node.key \), \( \text{INSERT}(node.left, x) \)
3. Otherwise, \( \text{INSERT}(node.right, x) \)

**SEARCH**\( (node, x) \)

1. If \( node \) is empty, return \(-\infty\)
2. If \( node.key == x \), return \( \text{size}(node.left) \)
3. If \( x < node.key \), return \( \text{SEARCH}(node.left, x) \)
4. If \( x > node.key \), return \( \text{SEARCH}(node.right, x) + 1 + \text{size}(node.left) \)
Key algorithms

**REMOVE**(*node, x*)

1. If `node.key == x`
   1. If the node is leaf, remove the node
   2. If the node only has left child, replace the current node to the left child
   3. If the node only has right child, replace the current node to the right child
   4. Otherwise, pick either maximum among left sub-tree or minimum among right subtree and substitute the node into the current node

2. If `x < node.key`
   1. Call **REMOVE**(*node.left, x*) if *node.left* exists
   2. Otherwise, return NOTFOUND

3. If `x > node.key`
   1. Call **REMOVE**(*node.right, x*) if *node.right* exists
   2. Otherwise, return NOTFOUND
Implementation of binary search tree

myTree.h

```cpp
template <class T>
class myTree {
protected:
    myTreeNode<T>* pRoot;  // tree contains pointer to root
    myTree(myTree& a) {};  // prevent copying

public:
    myTree() : pRoot(NULL) {}  // initially root is empty
    ~myTree() {
        if ( pRoot != NULL ) delete pRoot; }
    void insert(T x);
    int search(T x);
    bool remove(T x);
};
```
Implementation of binary search tree

myTreeNode.h

```cpp
template <class T>
class myTreeNode {
    T value;  // key value
    int size;  // total number of nodes in the subtree
    myTreeNode<T>* left;  // pointer to the left subtree
    myTreeNode<T>* right;  // pointer to the right subtree
    myTreeNode(T x, myTreeNode<T>* l, myTreeNode<T>* r); // constructors
    ~myTreeNode();       // destructors
    void insert(T x);   // insert an element
    int search(T x);
    myTreeNode<T>* remove(T x, myTreeNode<T>** ppSelf);
    T getMax();         // maximum value in the subtree
    T getMin();         // minimum value in the subtree
};
```
Recap

List

Tree

Hyun Min Kang

Biostatistics 615/815 - Lecture 7

September 27th, 2011

Summary

Binary search tree: Constructors and Destructors

myTreeNode.h

```
template<class T>
myTreeNode<T>::myTreeNode(T x, myTreeNode<T>* l, myTreeNode<T>* r) : value(x), size(1), left(l), right(r) {}

template<class T>
myTreeNode<T>::~myTreeNode() {
  // remove child nodes before removing the node itself
  if ( left != NULL ) delete left;
  if ( right != NULL ) delete right;
}
```
Binary search tree: **INSERT**

```cpp
// myTree.h

template <class T>
void myTree<T>::insert(T x) {
    if ( pRoot == NULL )
        pRoot = new myTreeNode<T>(x,NULL,NULL); // create a root if empty
    else
        pRoot->insert(x); // insert to the root
}
```
Binary search tree: **INSERT**

```cpp
template <class T>
void myTreeNode<T>::insert(T x) {
    if (x < value) { // if key is small, insert to the left subtree
        if (left == NULL)
            left = new myTreeNode<T>(x,NULL,NULL); // create if doesn't exist
        else
            left->insert(x);
    }
    else { // otherwise, insert to the right subtree
        if (right == NULL)
            right = new myTreeNode<T>(x,NULL,NULL);
        else
            right->insert(x);
    }
    ++size;
}
```
Binary search tree: **SEARCH**

```cpp
template <class T>
int myTree<T>::search(T x) {
    if ( pRoot == NULL )
        return -1;
    else
        return pRoot->search(x);
}
```
Binary search tree: **SEARCH**

```cpp
template<class T> // return the 0-based rank of the value x
int myTree<T>::search(T x) {
    if ( x == value ) { // if key matches to the value
        if ( left == NULL )
            return 0; // return 0 if there is no smaller element
        else
            return left->size; // return # of left-subtree otherwise
    }
    else if ( x < value ) { // recursively call the function to left subtree
        if ( left == NULL )
            return -1;
        else
            return left->search(x);
    }
}
```
MyTreeNode.h (cont'd)

```c
else { // when x > value, [#leftSubtree]+1 should be added
    if ( right == NULL )
        return -1;
    else {
        int r = right->search(x);
        if ( r < 0 ) return -1;
        else if ( left == NULL ) return ( 1 + r );
        else return ( left->size + 1 + r );
    }
}
```

Binary search tree: **SEARCH**
Binary search tree: **REMOVE**

```cpp
// myTree.h

template <class T>
bool myTree<T>::remove(T x) {
    if ( pRoot == NULL ) {
        return false;
    }
    else {
        myTreeNode<T>* p = pRoot->remove(x, pRoot);
        if ( p != NULL ) { // if an object was removed
            delete p;       // destroy the object
            return true;    // and return true
        }
        else {            // return false if the object was not found
            return false;
        }
    }
}
```
Binary search tree: **REMOVE**

### myTreeNode.h

```cpp
template <class T>
myTreeNode<T>* myTreeNode<T>::remove(T x, myTreeNode<T>*& pSelf) {
    if ( x == value ) { // key was found
        if ( ( left == NULL ) && ( right == NULL ) ) { // no child
            pSelf = NULL;
            return this;
        }
        else if ( left == NULL ) { // only left is NULL
            pSelf = right;
            right = NULL;
            return this;
        }
        else if ( right == NULL ) { // only right is NULL
            pSelf = left;
            left = NULL;
            return this;
        }
    } // ....
```
Binary search tree: **REMOVE** (cont’d)

```cpp
myTreeNode.h

else { // neither left nor right is NULL
    // choose which subtree to delete
    myTreeNode<T>* p;
    if ( left->size > right->size ) { // if left subtree is larger
        T m = left->getMax(); // copy the largest value among them
        p = left->remove(m, left); // to current node, and delete the node
        value = m;
    } else {
        T m = right->getMin(); // copy smallest value among them
        p = right->remove(m, right); // to current node, and delete the node
        value = m;
    }
    return p;
}

// ....
```
Binary search tree: **REMOVE** (cont’d)

```c
myTreeNode.h

else if ( x < value ) {
    if ( left == NULL )
        return NULL;
    else
        return left->remove(x, left);
}
else { // x > value
    if ( right == NULL )
        return NULL;
    else
        return right->remove(x, right);
}
```
Binary search tree: \texttt{getMax} and \texttt{getMin}

**myTreeNode.h**

```cpp
template <class T>
T myTreeNode<T>::getMax() {  // return the largest value
    if ( right == NULL ) return value;
    else return right->getMax();
}

template <class T>
T myTreeNode<T>::getMin() {  // return the smallest value
    if ( left == NULL ) return value;
    else return left->getMin();
}
```
If you want to print a tree...

```cpp
myTreeNode.h

template <class T> void myTreeNode<T>::print() {
    std::cout << "[ ";
    if ( left != NULL ) left->print();
    else std::cout << "[ NULL ]";
    std::cout << ", (" << value << "," << size << ") , ";
    if ( right != NULL ) right->print();
    else std::cout << "[ NULL ]";
    std::cout << " ]";
}

myTree.h

template <class T> void myTree<T>::print() {
    if ( pRoot != NULL ) pRoot->print();
    else std::cout << "(EMPTY TREE)";
    std::cout << std::endl;
}
```
Summary - Binary Search Tree

- Key Features
  - Fast insertion, search, and removal
  - Implementation is much more complicated
Summary - Binary Search Tree

- **Key Features**
  - Fast insertion, search, and removal
  - Implementation is much more complicated

- **Class Structure**
  - `myTree` class to keep the root node
  - `myTreeNode` class to store key and up to two children
Summary - Binary Search Tree

- Key Features
  - Fast insertion, search, and removal
  - Implementation is much more complicated

- Class Structure
  - myTree class to keep the root node
  - myTreeNode class to store key and up to two children

- Key Algorithms
  - **Insert**: Traverse the tree in sorted order and create a new node in the first leaf node.
  - **Search**: Divide-and-conquer algorithms
  - **Remove**: Move the nearest leaf element among the subtree and destroy it.
Two types of containers

Containers for single-valued objects - last lecture

- **INSERT**\((T, x)\) - Insert \(x\) to the container.
- **SEARCH**\((T, x)\) - Returns the location/index/existence of \(x\).
- **REMOVE**\((T, x)\) - Delete \(x\) from the container if exists
- STL examples include `std::vector`, `std::list`, `std::deque`, `std::set`, and `std::multiset`.

Containers for (key,value) pairs - this lecture

- **INSERT**\((T, x)\) - Insert \((x.key, x.value)\) to the container.
- **SEARCH**\((T, k)\) - Returns the value associated with key \(k\).
- **REMOVE**\((T, x)\) - Delete element \(x\) from the container if exits
- Examples include `std::map`, `std::multimap`, and `__gnu_cxx::hash_map`
Direct address tables

An example (key,value) container
- \( U = \{0, 1, \cdots, N - 1\} \) is possible values of keys (\( N \) is not huge)
- No two elements have the same key

Direct address table: a constant-time container
Let \( T[0, \cdots, N - 1] \) be an array space that can contain \( N \) objects
- \textsc{Insert}(\( T, x \)) : \( T[x\.key] = x \)
- \textsc{Search}(\( T, k \)) : \text{return } T[k]
- \textsc{Remove}(\( T, x \)) : \( T[x\.key] = \text{Nil} \)
Analysis of direct address tables

**Time complexity**

- Requires a single memory access for each operation
- $O(1)$ - constant time complexity

**Memory requirement**

- Requires to pre-allocate memory space for any possible input value
- $2^{32} = 4\text{GB} \times \text{(size of data)}$ for 4 bytes (32 bit) key
- $2^{64} = 18\text{EB} \times (1.8 \times 10^7 \text{TB}) \times \text{(size of data)}$ for 8 bytes (64 bit) key
- An infinite amount of memory space needed for storing a set of arbitrary-length strings (or exponential to the length of the string)
Key features

- \( O(1) \) complexity for \textbf{INSERT}, \textbf{SEARCH}, and \textbf{REMOVE}
- Requires large memory space than the actual content for maintaining good performance
- But uses much smaller memory than direct-address tables
Hash Tables

Key features

- $O(1)$ complexity for **INSERT**, **SEARCH**, and **REMOVE**
- Requires large memory space than the actual content for maintaining good performance
- But uses much smaller memory than direct-address tables

Key components

- Hash function
  - $h(x.key)$ mapping key onto smaller ‘addressible’ space $H$
  - Total required memory is the possible number of hash values
  - Good hash function minimize the possibility of key collisions
- Collision-resolution strategy, when $h(k_1) = h(k_2)$. 
Summary

Today

- List
- Binary Search Tree
- Direct Address Table
- Introduction to hash table

Next Lecture

- More hash tables
- Dynamic programming