Biostatistics 615/815 Lecture 7: Elementary Data Structures

Hyun Min Kang

September 25th, 2012

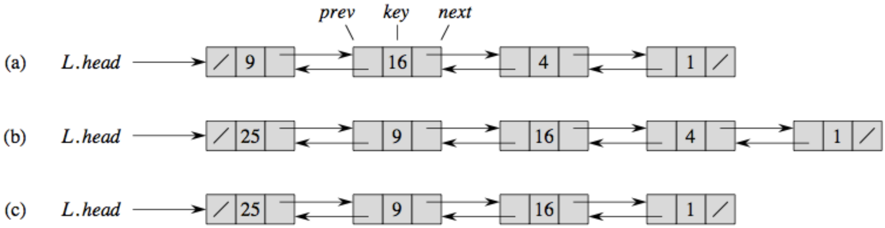
Simple Array

- Simplest container
- Constant time for insertion
- $\Theta(n)$ for search
- $\Theta(n)$ for remove
- Elements are clustered in memory, so faster than list in practice.
- Limited by the allocation size. $\Theta(n)$ needed for expansion

Sorted Array

- $\Theta(n)$ for insertion
- $\Theta(\log n)$ for search
- $\Theta(n)$ for remove
- Optimal for frequent searches and infrequent updates
- Limited by the allocation size. $\Theta(n)$ needed for expansion

Linked list



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

Implementation of singly-linked list

myList.h

```
#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(const T& x); // insert an element x
  bool search(const T& x);  // search for an element x and return its location
  bool remove(const T& x); // delete a particular element
  void print();  // print the content of array to the screen
};
```

List implementation : `class myListNode`

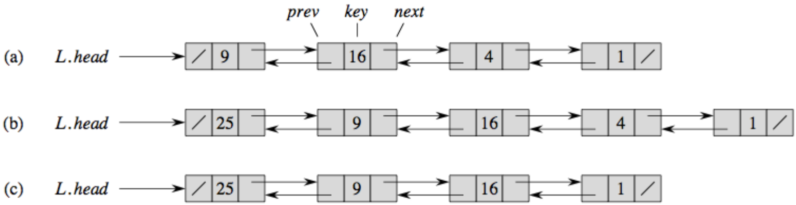
myListNode.h

```
// 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(const T& x, myListNode<T>* n) : value(x), next(n) {} // constructor
  ~myListNode();
  bool search(const T& x);
  myListNode<T>* remove(const T& x, myListNode<T>*& prevNext);
  void print(char c);
  template <class S> friend class myList; // allow full access to myList class
};
```

Inserting an element to a list

myList.h

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



Destructor is required because new was used

myList.h

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

myListNode.cpp

```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

myList.h

```
template <class T>
bool myList<T>::search(const T& x) {
  if ( head == NULL )  return false;  // NOT_FOUND if empty
  else return head->search(x);  // search from the head node
}
```

myListNode.cpp

```
template <class T>
// search for element x, and the current index is curPos
bool myListNode<T>::search(const T& x) {
  if ( value == x )          return true; // if found return current index
  else if ( next == NULL ) return false;     // NOT_FOUND if reached end-of-list
  else return next->search(x);  // recursive call until terminates
}
```

Removing an element from a list

myList.h

```
template <class T>
bool myList<T>::remove(const 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

```
template <class T>
// pass the pointer to [prevElement->next] so that we can change it
myListNode<T>* myListNode<T>::remove(const 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

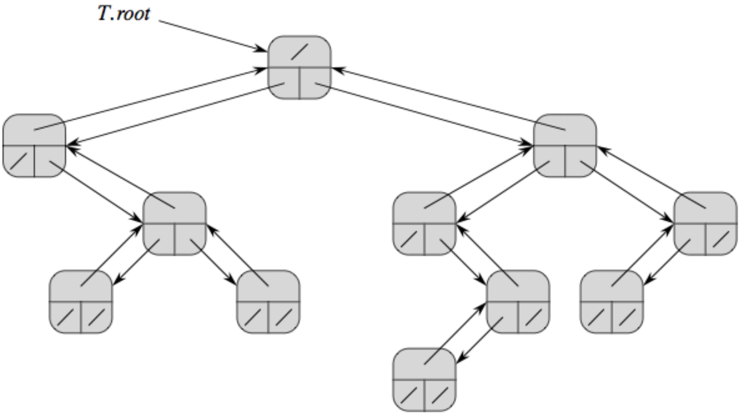
- Class Structure
  - `myList` class to keep the head node
  - `myListNode` class to store key and pointer to next node
- `Insert` algorithm : Create a new node as a head node
- `Search` algorithm
  - Return the index if key matches
  - Otherwise, advance to the next node
- `Remove` algorithm :
  - Search 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?

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 : `left.key` $\leq$ `key` $\leq$ `right.key`
- Average $\Theta(\log n)$ complexity for insert, search, remove operations

An example binary search tree



Key algorithms

$\textsc{Insert}(node, x)$

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

$\textsc{Search}(node, x)$

1. If $node$ is empty, return FALSE
2. If $node.key == x$, return TRUE
3. If $x < node.key$, return $\textsc{Search}(node.left, x)$
4. If $x > node.key$, return $\textsc{Search}(node.right, x)$

Implementation of binary search tree

myTree.h

```
#include <iostream>
#include "myTreeNode.h"

template <class T>
class myTree {
protected:
  myTreeNode<T> *pRoot;    // list only contains the pointer to head
  myTree(myTree& a) {};    // prevent copying
public:
  myTree() : pRoot(NULL) {} // initially header is NIL
  ~myTree() {}
  void insert(const T& x); // insert an element x
  bool search(const T& x);  // search for an element x and return its location
  bool remove(const T& x); // delete a particular element
  void print();
};
```

Implementation of binary search tree

myTreeNode.h

```
#include <iostream>
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(const T& x, myTreeNode<T>* l, myTreeNode<T>* r); // constructors
  ~myTreeNode();             // destructors
  void insert(const T& x); // insert an element
  bool search(const T& x);
  myTreeNode<T>* remove(const T& x, myTreeNode<T>*& pSelf);
  const T& getMax();                 // maximum value in the subtree
  const T& getMin();                 // minimum value in the subtree
  void print();
  template <class S> friend class myTree; // allow full access to myList class
};
```

Binary search tree : Constructors and Destructors

myTreeNode.h

```
template<class T>
myTreeNode<T>::myTreeNode(const 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

myTree.h

```
template <class T>
void myTree<T>::insert(const 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

myTreeNode.h

```
template <class T>
void myTreeNode<T>::insert(const 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

myTree.h

```
template <class T>
bool myTree<T>::search(const T& x) {
  if ( pRoot == NULL )
    return false;
  else
    return pRoot->search(x);
}
```

Binary search tree : Search

myTreeNode.h

```
template <class T>
bool myTreeNode<T>::search(const T& x) {
  if ( x == value ) {
      return true;
  }
  else if ( x < value ) {
    if ( left == NULL )
      return false;
    else
       return left->search(x);
  }
  else {
    if ( right == NULL )
      return false;
    else
       return right->search(x);
  }
}
```

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

- $\text{INSERT}(T, x)$ - Insert x to the container.
- $\text{SEARCH}(T, x)$ - Returns the location/index/existence of x.
- $\text{REMOVE}(T, x)$ - 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

- $\text{INSERT}(T, x)$ - Insert $(x.key, x.value)$ to the container.
- $\text{SEARCH}(T, k)$ - Returns the value associated with key k.
- $\text{REMOVE}(T, x)$ - Delete element x from the container if exitst
- 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 continaer

Let $T[0, \cdots, N-1]$ be an array space that can contain N objects

- $\textsc{Insert}(T, x)$: $T[x.key] = x$
- $\textsc{Search}(T, k)$: $\textsc{return}$ $T[k]$
- $\textsc{Remove}(T, x)$: $T[x.key] = \textsc{Nil}$

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
- $2^{32} = 4GB\times$(size of data) for 4 bytes (32 bit) key
- $2^{64} = 18EB(1.8 \times 10^7 TB)\times$(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)

Hash Tables

Key features

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

Hash Tables

Key features

- $O(1)$ complexity for INSERT, SEARCH, and REMOVE
- Requires large memory space than the actual content for maintainng good performance
- But uses much smaller memory than direct-addres 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 $h(k_1) = h(k_2)$.

Chained hash : A simple example

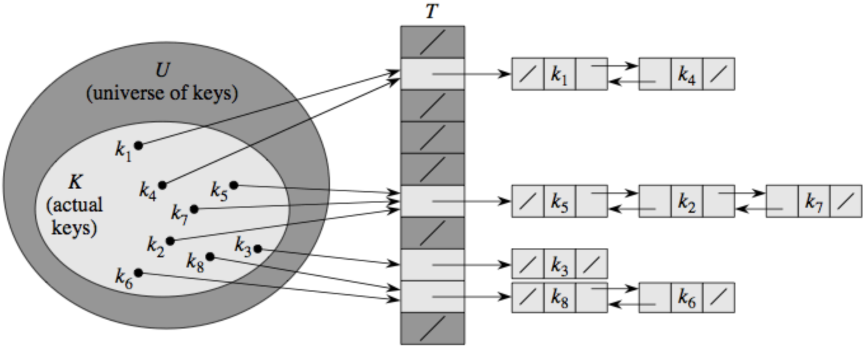
A good hash function

- Assume that we have a good hash function $h(x.key)$ that 'fairly uniformly' distribute key values to H
- What makes a good hash function will be discussed later today.

A `ChainedHash`

- Each possible hash key contains a linked list
- Each linked list is originally empty
- An input `(key,value)` pair is appened to the linked list when inserted
- $O(1)$ time complexity is guaranteed when no collision occurs
- When collision occurs, the time complexity is proportional to size of linked list assocated with $h(x.key)$

Illustration of ChainedHash



Simplfied algorithms on ChainedHash

Initialize(T)

- Allocate an array of list of size m as the number of possible key values

Insert(T, x)

- Insert x at the head of list $T[h(x.key)]$.

Search(T, k)

- Search for an element with key k in list $T[h(k)]$.

Remove(T, x)

- Delete x fom the list $T[h(x.key)]$.

Open Addressing

Chained Hash - Pros and Cons

- △ Easy to understand
- △ Behavior at collision is easy to track
- ▽ Every slots maintains pointer - extra memory consumption
- ▽ Inefficient to dereference pointers for each access
- ▽ Larger and unpredictable memory consumption

Open Addressing

Chained Hash - Pros and Cons

- △ Easy to understand
- △ Behavior at collision is easy to track
- ▽ Every slots maintains pointer - extra memory consumption
- ▽ Inefficient to dereference pointers for each access
- ▽ Larger and unpredictable memory consumption

Open Addressing

- Store all the elements within an array
- Resolve conflicts based on predefined probing rule.
- Avoid using pointers - faster and more memory efficient.
- Implementation of REMOVE can be very complicated

Hash tables : summary

- Linear-time performance container with larger storage
- Key components
  - Hash function
  - Conflict-resolution strategy
- Chained hash
  - Linked list for every possible key values
  - Large memory consumption + deferencing overhead
- Open Addressing
  - Probing strategy is important

When are binary search trees better than hash tables?

When are binary search trees better than hash tables?

- When the memory efficiency is more important than the search efficiency

When are binary search trees better than hash tables?

- When the memory efficiency is more important than the search efficiency
- When many input key values are not unique

When are binary search trees better than hash tables?

- When the memory efficiency is more important than the search efficiency
- When many input key values are not unique
- When querying by ranges or trying to find closest value.

Next Lecture

- Dynamic programming