Biostatistics 615/815 Lecture 7: Elementary Data Structures

Hyun Min Kang

September 25th, 2012

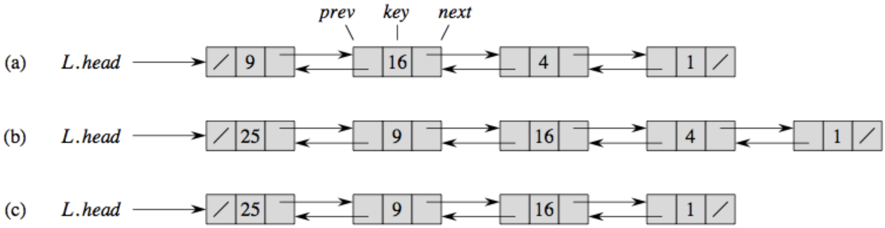
Simple Array

- 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`

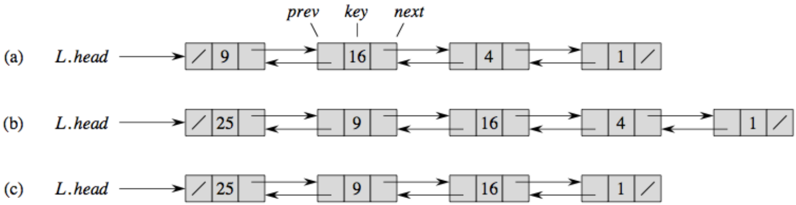
myListNode.h

```
// 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

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

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

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

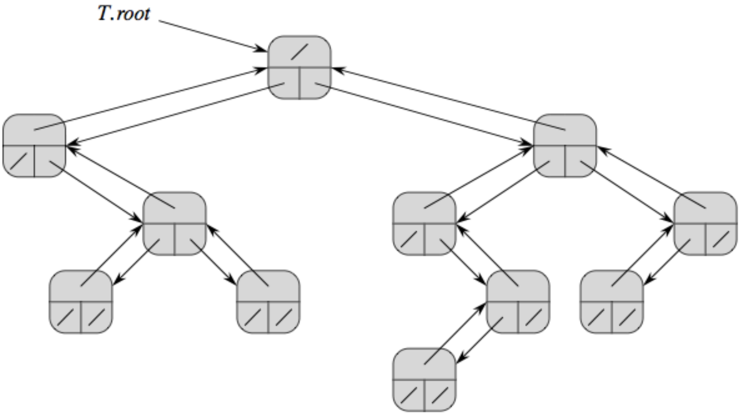
- Class Structure
    - `myList` class to keep the head node
    - `myListNode` class to store key and pointer to next node
- `Insert` algorithm : Create a new node as a head node
- `Search` algorithm
    - Return the index if key matches
    - Otherwise, advance to the next node
- `Remove` algorithm :
    - 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 $-\infty$
2. If $node.key == x$, return size$(node.left)$
3. If $x < node.key$, return $\textsc{Search}(node.left, x)$
4. If $x > node.key$, return $\textsc{Search}(node.right, x) + 1 +$ size$(node.left)$

Key algorithms

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

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 $\textsc{Remove}(node.left, x)$ if $node.left$ exists
   2. Otherwise, return NOTFOUND
3. If $x > node.key$
   1. Call $\textsc{Remove}(node.right, x)$ if $node.right$ exists
   2. Otherwise, return NOTFOUND

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 ) {          // if key matches to the value
    if ( left == NULL ) return true;
    else return true;
  }
  else if ( x < value ) {      // recursively call the function to left subtree
    if ( left == NULL ) return false;
    else return left->search(x);
  }
  else {
    if ( right == NULL ) return false;
    else {
      int r = right->search(x);
      if ( r < 0 ) return false;
      else if ( left == NULL ) return true;
      else return true;
    }
  }
}
```

Binary search tree : REMOVE

myTree.h

```
template <class T>
bool myTree<T>::remove(const 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;     // return false if the object was not found
    }
  }
}
```

Binary search tree : REMOVE

myTreeNode.h

```
template <class T>
myTreeNode<T>* myTreeNode<T>::remove(const 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)

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
      const 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 {
      const 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)

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 : GETMAX and GETMIN

myTreeNode.h

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

template <class T>
const T& myTreeNode<T>::getMin() {  // return the smallest value
  if ( left == NULL ) return value;
  else return left->getMin();
}
```

If you want to print a tree...

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

- $\textsc{Insert}(T, x)$ - Insert x to the container.
- $\textsc{Search}(T, x)$ - Returns the location/index/existence of x.
- $\textsc{Remove}(T, x)$ - 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

- $\textsc{Insert}(T, x)$ - Insert $(x.key, x.value)$ to the container.
- $\textsc{Search}(T, k)$ - Returns the value associated with key k.
- $\textsc{Remove}(T, x)$ - 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

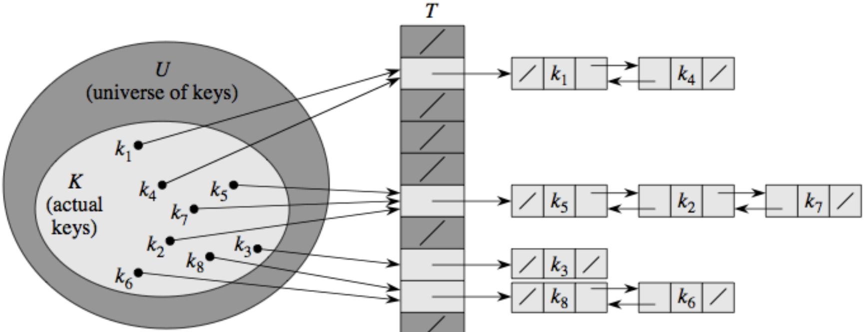
A good hash function

- 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 $T[h(x.key)]$.

Analysis of hashing with chaining

Assumptions

- Simple uniform hashing
  - $\Pr(h(k_1) = h(k_2)) = 1/m$ input key pairs k_1 and k_2.
- n is the number of elements stores
- Load factor $\alpha = n/m$.

Expected time complexity for SEARCH

- $X_{ij} \in \{0, 1\}$ a random variable of key collision between x_i and x_j.
- $E[X_{ij}] = 1/m$.

$$T(n) = \frac{1}{n} E\left[\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n}(X_{ij})\right)\right] = \Theta(1+\alpha)$$

Interesting properties (under uniform hash)

Probability of an empty slot

$$\Pr(k_1 \neq k, k_2 \neq k, \cdots, k_n \neq k) = \left(1 - \frac{1}{m}\right)^n \approx e^{-\alpha}$$

Birthday paradox : expected # of elements before the first collision

$$Q(H) \approx \sqrt{\frac{\pi}{2} m}$$

Coupon collector problem : expect # of elements to fill every slot

$$\sum_{i=1}^{m} \frac{m}{i} \approx m(\ln m + 0.577)$$

Hash functions

Making a good hash functions

- A hash function $h(k)$ is a deterministic function from $k \in K$ onto $h(k) \in H$.
- A good hash function distributes map the keys to hash values as uniform as possible
- The uniformity of hash function should not be affected by the pattern of input sequences

Example hash functions

- $k \in [0, 1)$, $h(k) = \lfloor km \rfloor$
- $k \in \mathbb{N}$, $h(k) = k \mod m$

'Good' and 'bad' hash functions

An example : $h(k) = \lfloor km \rfloor$

- When the input if uniformly distributed
  - $h(k)$ is uniformly distributed between 0 and $m-1$.
  - $h(k)$ is a good hash function
- When the input is skewed : $\Pr(k < 1/m) = 0.9$
  - More than 80% of input key pairs will have collisions
  - $h(k)$ is a bad hash function
  - Time complexity is close to a single linked list

Good hash functions

- 'Goodness' of a hash function can be dependent on the data
- If it is hard to create adversary inputs to make the hash function 'bad', it is generally a good hash function.

Examples of good hash functions

For integers

- Make the hash size m to be a large prime
- $h(k) = k \mod m$

For floating point values $k \in [0, 1)$

- Make the hash size m to be a large prime
- $h(k) = \lfloor k * N \rfloor \mod m$ for a large number N.

For strings

- Pretend the string is a number with numeral system of $|\Sigma|$, where Σ is the set of possible characters.
- Apply the same hash function for integers

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

Probing in open hash

Modified hash functions

- $h : K \times H \rightarrow H$
- For every $k \in K$, the probe sequence $< h(k, 0), h(k, 1), \cdots, h(k, m-1) >$ must be a permutation of $< 0, 1, \cdots, m-1 >$.

Algorithm OPENHASHINSERT

```
Data: $T$ : hash, $k$ : key value to insert
Result: $k$ is inserted to $T$
for $i = 0$ to $m - 1$ do
    $j = h(k, i)$ if $T[j]$ ==NIL then
        $T[j] = k$;
        return $j$;
    end
end
error "hash table overflow";
```

Algorithm OpenHashSearch

```
Data: $T$ : hash, $k$ : key value to search
Result: Return $T[k]$ if exist, otherwise return Nil
for $i = 0$ to $m - 1$ do
    $j = h(k, i)$;
    if $T[j] == k$ then
        return $j$;
    end
    else if $T[j] ==$ Nil then
        return Nil;
    end
end
return Nil;
```

Strategies for Probing

Linear Probing

- $h(k,i) = (h'(k) + i) \mod m$
- Easy to implement
- Suffer from primary clustering, increasing the average search time

Strategies for Probing

Linear Probing

- $h(k,i) = (h'(k) + i) \mod m$
- Easy to implement
- Suffer from primary clustering, increasing the average search time

Quadratic Probing

- $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \mod m$
- Beter than linear probing
- Seconary clustering : $h(k_1, 0) = h(k_2, 0)$ implies $h(k_1, i) = k(k_2, i)$

Strategies for Probing

Double Hashing

- $h(k, i) = (h_1(k) + ih_2(k)) \mod m$
- The probe sequence depends in two ways upon k.
- For example, $h_1(k) = k \mod m$, $h_2(k) = 1 + (k \mod m')$
- Avoid clustering problem
- Performance close to ideal scheme of uniform hashing.

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
    - Double hashing is close to ideal hashing

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