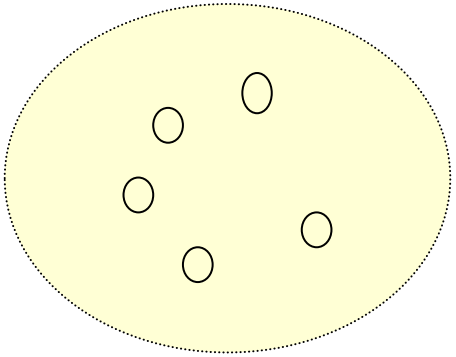
中山大學
SUN YAT-SEN UNIVERSITY

Lecture 2
Recap of Data Structures

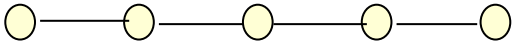
Algorithm Design

zhangzizhen@gmail.com

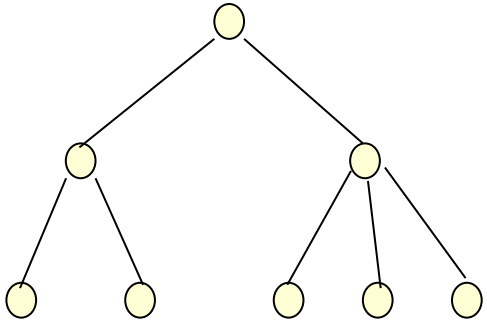
Classification of data structures



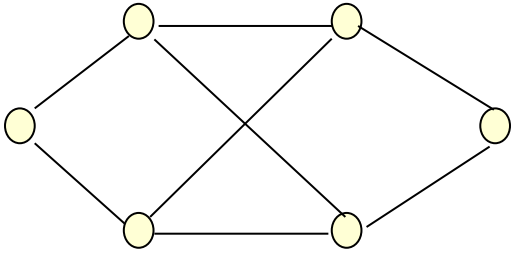
(a) set



(b) linear



(c) tree



(d) graph

List of data structures

- Data type
  - Primitive types: Boolean, Integer, Double, Character, Enum, etc.
  - Composite types: Array, Struct, Union, etc.
  - Abstract data types: Stack, List, Map, Set, String, Queue, etc.
- Linear data structures
  - Arrays
  - Lists
- Trees
  - Binary trees
  - Heap
  - Multiway trees
  - …
- Graphs

Abstract Data Type (ADT)

- An Abstract Data Type (ADT) is a data type that is organized in such a way that the specification on the objects and specification of the operations on the objects are separated from the representation of the objects and the implementation on the operations.
- 用三元组描述如下：

  ADT =（D,S,P）

其中：D是数据对象，S是D上的关系集，P是D的基本操作集。

ADT 抽象数据类型名{

数据对象：〈数据对象的定义〉

数据关系：〈数据关系的定义〉

基本操作：〈基本操作的定义〉

} ADT 抽象数据类型名

Stacks

- A stack is a data structure in which all insertions and deletions of entries are made at one end, called the top of the stack. The last entry which is inserted is the first one that will be removed. The operations perform in a Last-In-First-Out (LIFO) manner.
- Implementing a Contiguous Stack
  - We set up an array to hold the entries in the stack and a counter to indicate how many entries there are.
- Implementing a Linked Stack
  - A linked structure is made up of nodes, each containing both the information that is to be stored as an entry of the structure and a pointer telling where to find the next node in the structure.

Class Declaration for Stacks

```
const int maxstack = 10;

class Stack {
  public:
  Stack( );
  bool empty( ) const;
  Error_code pop( );
  Error_code top(Stack entry &item) const;
  Error_code push(const Stack_entry &item);
private:
  int count;
  Stack_entry entry[maxstack];
};
```

Contiguous Stack Implementation

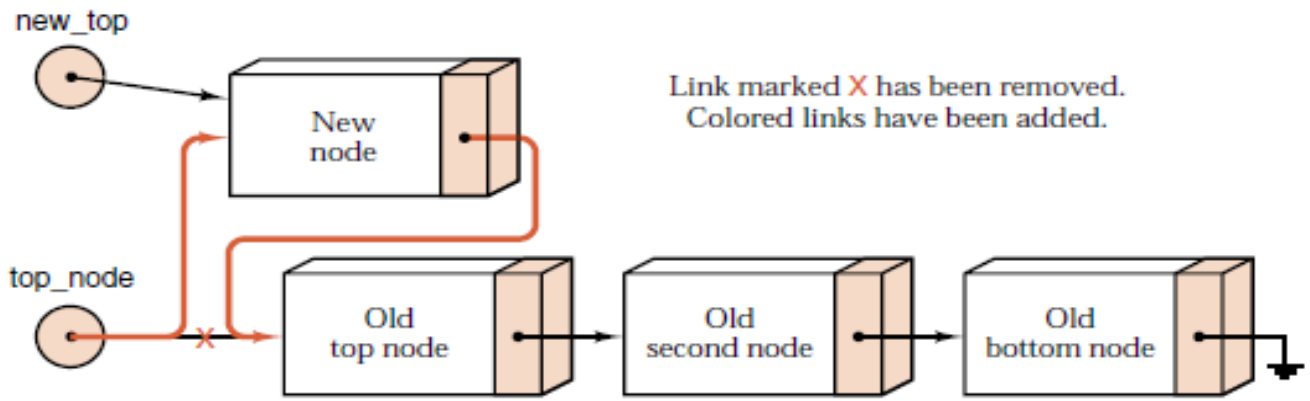
```
class Stack {
public:
  Stack( );
  bool empty( ) const;
  Error_code push(const Stack_entry &item);
  Error_code pop( );
  Error_code top(Stack_entry &item) const;
  // Safety features for linked structures
  ~Stack( );
  Stack(const Stack &original);
  void operator = (const Stack &original);
protected:
  Node *top_node;
};
```

Linked Stack Implementation

Linked Stack Operations

- **Pushing a Linked Stack**

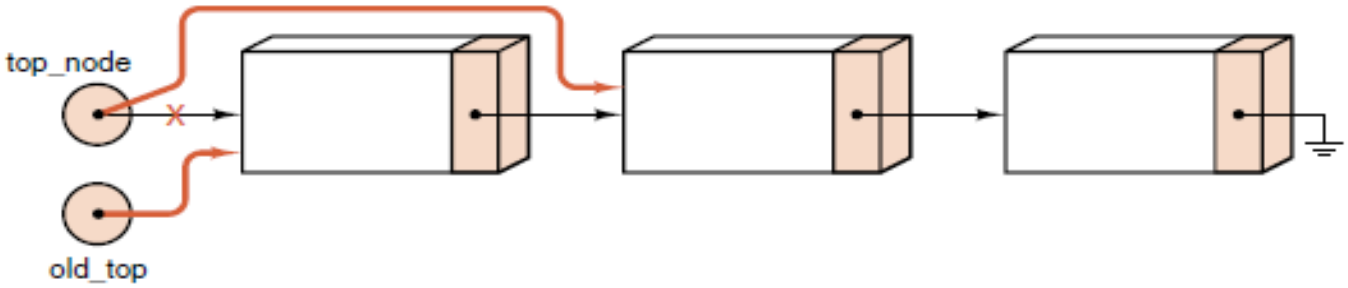
```
Error_code Stack :: push(const Stack_entry &item)
/* Post: Stack entry item is added to the top of the Stack; returns success
or returns a code of overflow if dynamic memory is exhausted. */
{
        Node *new_top = new Node(item, top_node);
        if (new_top == NULL) return overflow;
        top_node = new_top;
        return success;
}
```



Linked Stack Operations

- **Popping a Linked Stack**

```
Error_code Stack :: pop( )
/* Post: The top of the Stack is removed. If the Stack is empty the
method returns underflow; otherwise it returns success. */
{
        Node *old_top = top_node;
        if (top_node == NULL) return underflow;
        top_node = old_top->next;
        delete old_top;
        return success;
}
```



Queues

- **Definition**: A queue is a list in which all additions to the list are made at one end, and all deletions from the list are made at the other end. Queues are also called ***first-in, first-out lists,*** or ***FIFO*** for short***.***
- The first entry that will be removed from the queue, is called the *front* of the queue (or, sometimes, the *head* of the queue).
- The last entry in the queue, that is, the one most recently added, is called the *rear* (or the *tail*) of the queue.

Class Declaration for Queues

```
const int maxqueue = 10;

class Queue {
public:
  Queue( );
  bool empty( ) const;
  Error_code serve( );
  Error_code append(const Queue_entry
&item);
  Error_code retrieve(Queue_entry &item)
const;
protected:
  int count;
  int front, rear;
  Queue_entry entry[maxqueue];
};
```

Contiguous Stack Implementation

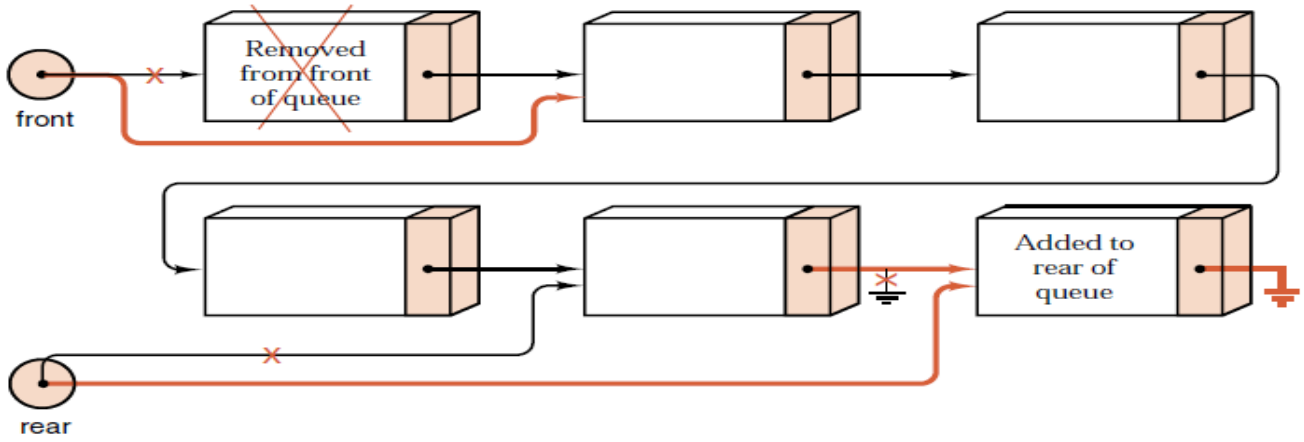
```
class Queue {
public:
  // standard Queue methods
  Queue( );
  bool empty( ) const;
  Error_code append(const Queue_entry &item);
  Error_code serve( );
  Error_code retrieve(Queue entry &item) const;
  // safety features for linked structures
  ~Queue( );
  Queue(const Queue &original);
  void operator = (const Queue &original);
protected:
  Node *front, *rear;
};
```

Linked Stack Implementation

Linked Queue Methods

Append an entry:

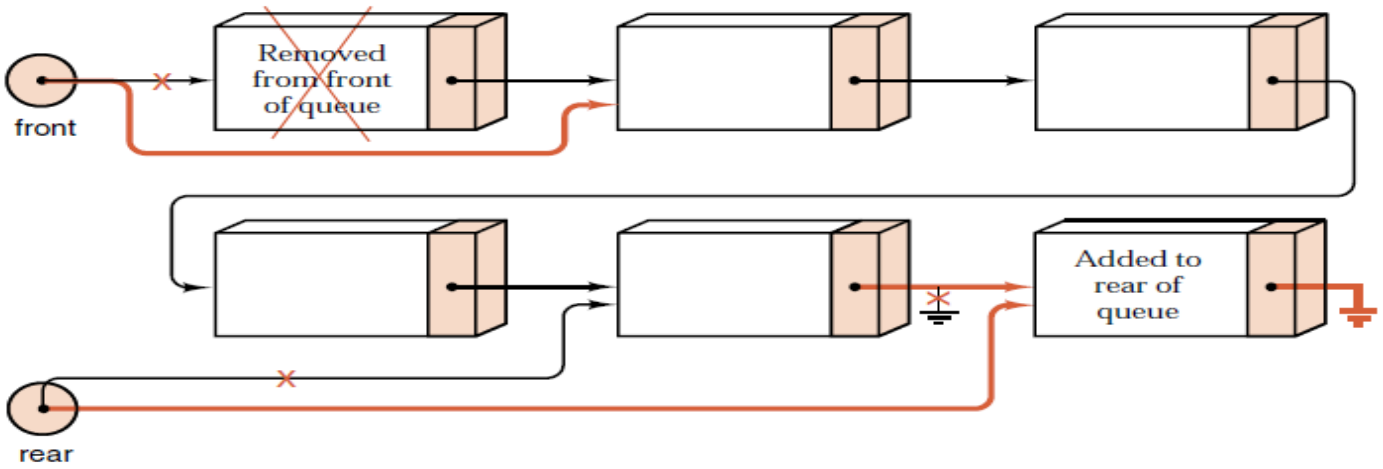
```
Error_code Queue :: append(const Queue_entry &item)
/* Post: Add item to the rear of the Queue and return a code of success or return a
code of overflow if dynamic memory is exhausted. */
{
        Node *new_rear = new Node(item);
        if (new_rear == NULL) return overflow;
        if (rear == NULL) front = rear = new_rear;
        else {
                rear->next = new_rear;
                rear = new_rear;
        }
        return success;
}
```



Linked Queue Methods

Serve an entry:

```
Error_code Queue :: serve( )
/* Post: The front of the Queue is removed. If the Queue is empty, return an Error
code of underflow. */
{
        if (front == NULL) return underflow;
        Node *old_front = front;
        front = old_front->next;
        if (front == NULL) rear = NULL;
        delete old_front;
        return success;
}
```



Circular implementation of queues

```
Queue :: Queue( )
/* Post: The Queue is initialized to be empty. */
{
    count = 0;
    rear = maxqueue - 1;
    front = 0;
}
Error_code Queue :: append(const Queue_entry &item)
/* Post: item is added to the rear of the Queue. If the Queue is full return
an Error_code of overflow and leave the Queue unchanged. */
{
    if (count >= maxqueue) return overflow;
    count++;
    rear = ((rear + 1) == maxqueue) ? 0 : (rear + 1);
    entry[rear] = item;
    return success;
}
Error_code Queue :: serve( )
/* Post: The front of the Queue is removed. If the Queue is empty return an
Error code of underflow. */
{
    if (count <= 0) return underflow;
    count--;
    front = ((front + 1) == maxqueue) ? 0 : (front + 1);
    return success;
}
```

Lists and Strings

- A **list** of elements of type T is a finite sequence of elements of T together with the following operations:
  1. **Construct** the list, leaving it empty.
  2. Determine whether the list is **empty** or not.
  3. Determine whether the list is **full** or not.
  4. Find the **size** of the list.
  5. **Clear** the list to make it empty.
  6. **Insert** an entry at a specified position of the list.
  7. **Remove** an entry from a specified position in the list.
  8. **Retrieve** the entry from a specified position in the list.
  9. **Replace** the entry at a specified position in the list.
  10. **Traverse** the list, performing a given operation on each entry.

Searching

- Sequential search
  - The method is simple;
  - Efficient for short list;
  - The list don’t have to be ordered;
  - Disaster for long list to be searched;
- Binary search
  - In searching an ordered list, first compare the target to the key in the center of the list. If it is smaller, restrict the search to the left half; otherwise restrict the search to the right half, and repeat. In this way, at each step we reduce the length of the list to be searched by half.

Binary Search Version 1

```
Error_code binary_search_1 (const Ordered_list &the_list,
        const Key &target, int &position)
/* Post: If a Record in the list has Key equal to target, then position
locates one such entry and a code of success is returned. Otherwise,
not present is returned and position is undefined.
Uses: Methods for classes List and Record. */
{
    Record data;
    int bottom = 0, top = the_list.size( ) - 1;
    while (bottom < top) {
        int mid = (bottom + top) / 2;
        the_list.retrieve(mid, data);
        if (data < target)
            bottom = mid + 1;
        else top = mid;
    }
    if (top < bottom) return not_present;
    else {
        position = bottom;
        the_list.retrieve(bottom, data);
        if (data == target) return success;
        else return not_present;
    }
}
```

When the middle part of the list is reduced to size 1, it will be guaranteed to be the *first* occurrence of the target if it appears more than once in the list.

Binary Search Version 2

```
Error_code binary_search_2(const Ordered_list &the_list,
    const Key &target, int &position)
/* Post: If a Record in the list has key equal to target, then position
locates one such entry and a code of success is returned. Otherwise,
not present is returned and position is undefined.
Uses: Methods for classes Ordered_list and Record. */
{
    Record data;
    int bottom = 0, top = the_list.size( ) - 1;
    while (bottom <= top) {
        position = (bottom + top) / 2;
        the_list.retrieve(position, data);
        if (data == target) return success;
        if (data < target) bottom = position + 1;
        else top = position - 1;
    }
    return not_present;
}
```

Sorting

Name	Average	Worst	Memory	Stable
Bubble sort	n^2	n^2	*1*	Yes
Selection sort	n^2	n^2	*1*	Depends
Shell sort		$n^{1.25}$	*1*	No
Insert sort	n^2	n^2	*1*	Yes
Heapsort	*nlogn*	*nlogn*	*1*	No
Merge sort	*nlogn*	*nlogn*	*n*	Yes
Quick sort	*nlogn*	n^2	*logn*	Depends

Hashing

- Idea of Hashing:
  - A table can be indexed by a key.
  - If there is a function ***f*: *K* → *N***, where *K* is the set of possible keys and *N* is the set of indices of an array, then we can store an entry with key *K* at the location *f(K)* and hence access a table entry becomes easy.
- To use hashing we must
  - (a) find good hash functions
  - (b) determine how to resolve collisions.
- Hashing methods
  - Modular arithmetic
  - Folding
  - …

Hashing

- To deal with collisions, there are generally two solutions:

  (1) Open addressing（开发地址）
  - Linear Probing (线性探查): it starts with the hash address and searches sequentially for the target key or an empty position.
  - Quadratic probing (二次探查): if there is a collision at hash address h, probes the table at locations $h + 1$, $h + 2^2$, $h + 3^2$, …, that is, at locations $h + i^2$ (mod hashsize) for $i = 1,2,\ldots$

  (2) Chaining （链地址法或拉链法）: When there is a collision, the synonym (同义) is put into a linked list of synonyms.
- The load factor of a table is $\lambda = n/t$, n is the number of entries in the table and t the size of the array.
- λ can be seen as the average length of the chains.

Binary Trees

DEFINITION A ***binary tree*** is either empty, or it consists of a node called the ***root*** together with two binary trees called the ***left subtree*** and the ***right subtree*** of the root.

- Traversal of Binary Trees
  - Inorder
  - Preorder
  - Postorder
  - Level-by-level

Binary Search Trees

DEFINITION A ***binary search tree*** is a binary tree that is either empty or in which the data entry of every node has a key and satisfies the conditions:

1. The key of the left child of a node (if it exists) is less than the key of its parent node.
2. The key of the right child of a node (if it exists) is greater than the key of its parent node.
3. The left and right subtrees of the root are again binary search trees.

- Tree search
- Tree sort
- Tree node insertion
- Tree node removal

AVL Trees

- An *AVL* tree is a binary search tree in which the heights of the left and right subtrees of the root differ by at most 1 and in which the left and right subtrees are again AVL trees.
- With each node of an AVL tree is associated a *balance factor*（平衡因子，左子树的高度减去右子树的高度）that is *left higher*, *equal*, or *right higher* according, respectively, as the left subtree has height greater than, equal to, or less than that of the right subtree.
- Rotation
  - RR
  - LL
  - RL
  - LR

General Trees and Forests

- Tree Representation
  - Parent method
  - Multiple links
  - Child-link
  - First child next sibling
- Trie
- Forest Traverse
- Disjoint set
- Huffman Tree

Graphs

- Graph Representation
  - Adjacency Matrix (邻接矩阵)
  - Adjacency Lists (邻接表)
- Graph Traversal
  - Depth-first: It is roughly analogous to preorder traversal of an ordered tree.
  - Breadth-first: It is roughly analogous to level-by-level traversal of an ordered tree. It can be done by using a queue. The head is the vertex to be visited. When it is visited, all its adjacent vertices are put in the queue.

Graphs

- Topological Sorting
  - Let *G* be a directed graph with no cycles (DAG). A topological order for *G* is a sequential listing of all the vertices in *G* such that, for all vertices $v,w \in G$, if there is an edge from *v* to *w*, then *v* precedes *w* in the sequential listing.
- Topological Sorting Algorithms
  - Bread-first algorithm
  - Depth-first algorithm

Graphs

- Minimum Spanning Trees

> DEFINITION A ***minimal spanning tree*** of a connected network is a spanning tree such that the sum of the weights of its edges is as small as possible.

- Prim's algorithm

```
for (int i = 1; i < count; i++) {
   Vertex v;                          //   Add one vertex v to X on each pass.
   Weight min = infinity;
   for (w = 0; w < count; w++) if (!component[w])
      if (distance[w] < min) {
         v = w;
         min = distance[w]; }
   if (min < infinity) {
      component[v] = true;
      tree.add_edge(v, neighbor[v], distance[v]);
      for (w = 0; w < count; w++) if (!component[w])
         if (adjacency[v][w] < distance[w]) {
            distance[w] = adjacency[v][w];
            neighbor[w] = v;                    }
   }
   else break;                        //   finished a component in disconnected graph
}
```

- Kruskal's algorithm (union-find algorithm)

Graphs

- Shortest Path
  - Dijkstra's algorithm (single source)

```
found[source] = true;        //  Initialize with vertex source alone in the set S.
distance[source] = 0;
for (int i = 0; i < count; i++) {  //  Add one vertex v to S on each pass.
   Weight min = infinity;
   for (w = 0; w < count; w++) if (!found[w])
      if (distance[w] < min) {
         v = w;
         min = distance[w];
      }
   found[v] = true;
   for (w = 0; w < count; w++) if (!found[w])
      if (min + adjacency[v][w] < distance[w])
         distance[w] = min + adjacency[v][w];
  }
}
```

  - Bellman-ford (single source, detect negative-weight cycle)
  - Floyd-Warshall (all-pairs)

```
for k ← 1 to n do
  for i ← 1 to n do
    for j ← 1 to n do
      if (D_{i,k} + D_{k,j} < D_{i,j}) then
        D_{i,j} ← D_{i,k} + D_{k,j};
```

Thank you!