Data structures and Algorithms
Basic Data structures

Pham Quang Dung

Hanoi, 2012

Outline

1. Basic concepts

2. Array

3. Lists

4. Stacks

5. Queues

Basic concepts

- Data Types
    - set of values
    - data representation
    - set of operations
- Abstract Data Types (ADT)
    - set of values
    - set of operations

Data Types: primitive data types in C

- **Built-in data types In C programming language**

Type	Bits	Minimum value	Maximum value
byte	8	-128	127
short	16	-32768	32767
char	16	0	65535
int	32	-2147483648 $= -2^{31}$	3247483647 $= 2^{31} - 1$
long	64	-9223372036854775808	9223372036854775807
float	32		
double	64		

- Operations on primitive data types: $+$, -, *, /, ...

ADT

ADT	Object	Operations
List	nodes	insert, remove, find,...
Graphs	nodes, edges	findPath, Search,...
Stack	elements	push, pop, isEmpty,...
Queue	elements	enqueue, dequeue, isEmpty,...
Binary tree	nodes	traversal, find, ...

Outline

1 Basic concepts

2 Array

3 Lists

4 Stacks

5 Queues

Array

- Collection of elements of similar data type
- Elements are stored sequentially
- Each elements of the array is accessed via its index
- An array can have one or more dimensions

Example

```
int a[1000];
int b[100][100];
typedef struct MyStruct{
  int value;
  MyStruct* ptr;
};
MyStruct x[10];
```

Outline

1 Basic concepts

2 Array

3 Lists

4 Stacks

5 Queues

Lists

- List ADT implements an **ordered** collection of values (each value can appear several times)
- Notations
  - L: list of objects
  - x: an object
  - p: position type
  - $END(L)$: function that returns the position after the position of the last element of the list
- Operations
  - Insert(x, p, L): insert element x at position p of the list L
  - Locate(x, L): return the position of x in L
  - Retrieve(p, L): return the element at position p in L
  - Delete(p, L): remove element at position p in L
  - Next(p, L): return the position after the position p in L
  - Prev(p, L): return the position before the position p in L
  - MakeNull(L): set L to empty list and return $END(L)$
  - First(L): return the first position in L
  - PrintList(L): print all elements of L in the order they appear in L

Lists

- List ADT implements an **ordered** collection of values (each value can appear several times)
- Notations
  - L: list of objects
  - x: an object
  - p: position type
  - $END(L)$: function that returns the position after the position of the last element of the list
- Operations
  - Insert(x, p, L): insert element x at position p of the list L
  - Locate(x, L): return the position of x in L
  - Retrieve(p, L): return the element at position p in L
  - Delete(p, L): remove element at position p in L
  - Next(p, L): return the position after the position p in L
  - Prev(p, L): return the position before the position p in L
  - MakeNull(L): set L to empty list and return $END(L)$
  - First(L): return the first position in L
  - PrintList(L): print all elements of L in the order they appear in L

List - implementation

- Array-based
    - Elements located in contiguous blocks in memory
    - **Delete** and **Insert** operations are costly
- Pointer-based (linked list)
    - Collection of nodes that not necessarily locate in contiguous blocks
    - Single linked list: Each node contains the element (data) and a reference (pointer) to the next node
    - Doubly linked list: Each node contains the element (data), a reference to the previous node and a reference to the next node

Array-based implementation

Insertion and **Deletion**

```

int a[10000];
int n;// size of the list, elements are a[1], a[2], ..., a[n]
void insert(int x, int pos){
  for(int i = n; i >= pos; i--)
    a[i+1] = a[i];
  a[pos] = x;
  n = n + 1;
}

void del(int k){
  for(int i = k; i <= n-1; i++)
    a[i] = a[i+1];
  n = n - 1;
}
```

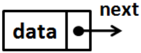
Array-based implementation

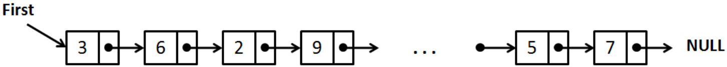
MakeNull and **PrintList**

```
void makeNull(){
  n = 0;
}
void printList(){
  for(int i = 1; i <= n; i++)
    printf("%d ",a[i]);
  pritnf("\n");
}
void retrieve(int k){
  return a[k];
}
int end(){
  return -1;
}
int locate(int x){
  for(int i = 1; i <= n; i++)
    if(a[i] == x)
      return i;
  return end();
}
```

Pointer-based implementation

Single linked list: representation





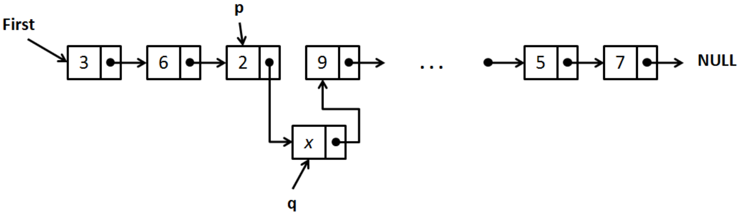
```
typedef int ElementType;

struct PointerType{
  ElementType data;
  PointerType* next;
};

PointerType* first = NULL;
```

Pointer-based implementation

Single linked list: Insertion



Pointer-based implementation

Single linked list: Insertion

```
PointerType* insertAfter(ElementType x, PointerType* p){
  // insert an element x into the position after p
  PointerType* q;
  q = new PointerType;
  q->data = x;

  if(first == NULL){
    q->next = NULL;
    first = q;
  }else{
    q->next = p->next;
    p->next = q;
  }

  return q;
}
```

Pointer-based implementation

Single linked list: Deletion

```
void del(PointerType* p){
  if(p == first){
    PointerType* tmp = first->next;
    delete first;
    first = tmp;
  }else{
    PointerType* pi = first;
    while(pi != NULL && pi->next != p)
      pi = pi->next;
    if(pi != NULL){
      pi->next = p->next;
      delete p;
    }
  }
}
```

Pointer-based implementation

Single linked list: **PrintList** and **MakeNull**

```
void printList(){
  PointerType* p = first;
  while(p != NULL){
    printf("%d ",p->data);
    p = p->next;
  }
  printf("\n");
}
PointerType* makeNull(){
  while(first != NULL){
    PointerType* tmp = first ->next;
    delete first;
    first = tmp;
  }
  return NULL;
}
```

Pointer-based implementation

Single linked list: Previous

```
PointerType* prev(PointerType* p){
  PointerType* tmp = first;
  while(tmp != NULL){
    if(tmp->next == p)
      return tmp;
    tmp = tmp->next;
  }
  return NULL;
}
```

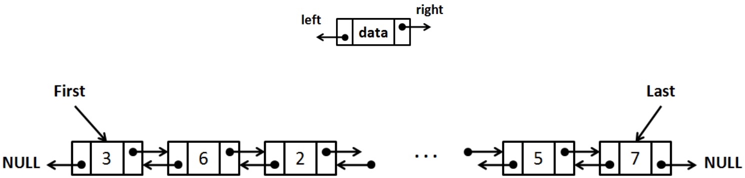
Pointer-based implementation

Single linked list: Locate

```
PointerType* locate(ElementType x){
  PointerType* p = first;
  while(p != NULL){
    if(p->data == x)
      return p;
    p = p->next;
  }
  return NULL;
}
```

Pointer-based implementation

Doubly linked list: representation

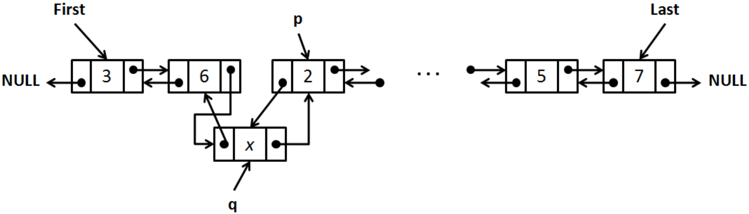


```
struct Node{
  int data;
  Node*  left;
  Node* right;
};

Node* first = NULL;
Node* last = NULL;
```

Pointer-based implementation

Doubly linked list: Insertion (insert an element at a position pointed by a pointer p)



Pointer-based implementation

Doubly linked list: Insertion (insert an element at a position pointed by a pointer p)

```
void insertAt(int x, Node* p){
  Node* q = new Node;
  q->right = p;
  q->data = x;
  Node* p1 = p->left;
  if(p1 != NULL)
    p1->right = q;
  q->left = p1;
  p->left = q;
}
```

Pointer-based implementation

Doubly linked list: Insertion (insert an element to the end of the list)

```
void insertToEnd(int x){
  Node* p = new Node;
  p->data = x;
  if(first == NULL){
    p->left = NULL;
    p->right = NULL;
    first = p;
    last = p;
  }else{
    p->right = NULL;
    p->left = last;
    last->right = p;
    last = p;
  }
}
```

Built-in List in C++

- Manual: http://www.cplusplus.com/reference/list/list/
- Fundamental methods
    - empty()
    - size()
    - push_front()
    - pop_front()
    - push_back()
    - pop_back()
    - insert()
    - erase()
    - clear()

Outline

1 Basic concepts

2 Array

3 Lists

4 Stacks

5 Queues

Stacks

- Stack ADT: An ordered list in which all insertions and deletions are made at one end (called top)
- Principle: the last element inserted into the stack must be the first one to be removed (**Last-In-First-Out**)
- Operations
  - Push(x, S): push an element x into the stack S
  - Pop(S): remove an element from the stack S, and return this element
  - Top(S): return the element at the top of the stack S
  - Empty(S): return true if the stack S is empty

Linked list-based implementation

```
struct Node{
  char info;
  Node* next;
};

Node* top = NULL;// pointer to the top of the stack


int stackEmpty(){
  if(top == NULL)
    return 1;
  else
    return 0;
}
```

Linked list-based implementation

```
void push(char x){
  Node* p;
  p = new Node;
  p->info = x;
  p->next = top;
  top = p;
}

char pop(){
  char x = top->info;
  Node* p = top;
  top = top->next;
  delete p;
  return x;
}
```

Application: Parentheses matching

- ()([]){}: consistent
- ()()[{): not consistent
- [](){[])[]: not consistent

Application: Parentheses matching

```
int checkMatch(char x, char y){
  if(x == '(' && y == ')') return 1;
  if(x == '[' && y == ']') return 1;
  if(x == '{' && y == '}') return 1;

  return 0;
}
```

Application: Parentheses matching

```
int check(char X[], int n){
  for(int i = 0; i < n; i++){
    if(X[i] == '(' || X[i] == '[' || X[i] == '{')
      push(X[i]);
    else{
      if(X[i] == ')' || X[i] == ']' || X[i] == '}'){
        if(stackEmpty()) return 0;
        else{
          char x = pop();
          if(checkMatch(x,X[i]) == 0) return 0;
        }
      }
    }
  }
  if(stackEmpty()) return 1; else return 0;
}
```

Outline

1. Basic concepts
2. Array
3. Lists
4. Stacks
5. Queues

Queues

- Queue ADT: An ordered list in which the insertions are made at one end (called tail) and the deletions are made at the other end (called head)
- Principle: the first element inserted into the queue must be the first one to be removed (**First-In-First-Out**)
- Applications: items do not have to be processed immediately but they have to be processed in FIFO order
  - Data packets are stored in a queue before being transmitted over the internet
  - Data is transferred asynchronously between two processes: IO buffered, piples, etc.
  - Printer queues, keystroke queues (as we type at the keyboard), etc.

Queues

- Operations
    - Enqueue(x, Q): push an element x into the queue Q
    - Dequeue(Q): remove an element from the queue Q, and return this element
    - Head(Q): return the element at the head of the queue Q
    - Tail(Q): return the element at the tail of the queue Q
    - Empty(Q): return true if the queue Q is empty

Built-in Stack and Queue in C++

- Stack (manual: http://www.cplusplus.com/reference/stack/stack/)
  - empty: Test whether the stack is empty
  - size: Return size
  - top: Access next element
  - push: Add element
  - pop: Remove element
- Queue (manual: http://www.cplusplus.com/reference/queue/queue/)
  - empty: Test whether the queue is empty
  - size: Return size
  - front: Access next element
  - back: Access last element
  - push: Insert element
  - pop: Delete next element

Queues: palindrome strings checking

- A palindrome string is the string that reads the same forward and backward
- Example: MADAM, NOON, RADAR, etc.
- Problem: check whether a given string is palindrome
    - Use queue and stack

Queues: palindrome strings checking

```
#include <stdio.h>
#include <queue>
#include <stack>
#include <string.h>
using namespace std;
int main(int argc, char** argv){
  char* s = argv[1];
  queue<char> Q;
  stack<char> S;
  for(int i = 0; i < strlen(s); i++){
    Q.push(s[i]);     S.push(s[i]);
  }
  while(!Q.empty()){
    if(Q.front() != S.top()){
      printf("not palindrome\n");
      return 0;
    }
    Q.pop();     S.pop();
  }
  printf("palindrome\n");
}
```