Ch6佇列

Queue — 1. ADT Queue Operation
2. Simple Queue Application
 - Reading a String of Characters
 - Recognize a Palindrome
3. Implementations of ADT Queue
4. Application of Queues — Simulation

06-1佇列的實作

1. New item enter at the back, or rear, of the queue
 Item leave from the front of the queue
 First-In, First-out (FIFO)
2. ex:

```
aQueue.createQueue()            ↙front
aQueue.enqueue(5)               5
aQueue.enqueue(2)               52
aQueue.enqueue(7)               527 ←back
aQueue.getFront(queueFront)     527  (queueFront is 5)
aQueue.dequeue(queueFront)      27   (queueFront is 5)
aQueue.dequeue(queueFront)      7    (queueFront is 2)
```

3. ex: 247=(2*10+4)*10+7

```
do{
  aQueue.dequeue(ch);        //空白,忽略
}while(ch is blank)
n=0;
done=FALSE;
while(!done and ch is digit){
  n=n*10+ integer represented by ch;
  if(aQueue.isEmpty())
    done=TRUE;
  else
    aQueue.dequeue(ch)
}
```

06-2佇列辨識迴文

1. Stack reverse the order of occurrences
 Queue preserves the order of occurrences
2. Insert character into both a queue and a stack
 Compare the characters at the front of the queue and the top of the stack

3、isPal(in str: string): boolean

```
aQueue.createQueue( );
aStack.creatStack( );
for(the next character ch in str){
  aQueue.enqueue(ch);
  aStack.push(ch);
}
charEqual = true;
while( !aQueue.isempty() && charEqual){
  aQueue.dequeue(front);
  astack.pop(top);
  if(front != top)
    charEqual=FALSE;
}
```

06-3 以指標實作佇列

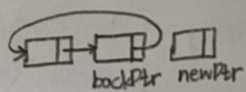
1、Linear-linked list — front, back
Circular-linked list — back

2、新增 enqueue — ① newPtr→next = NULL;
bockPtr→next = newPtr;
bockPtr = newPtr;
② frontPtr = newPtr;
bockPtr = newPtr;

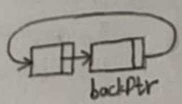
3、刪除 dequeue — ① tempPtr = frontPtr;
frontPtr = frontPtr→next;
tempPtr→next = NULL;
delete tempPtr;
② tempPtr = frontPtr
frontPtr = NULL;
bockPtr = NULL;
tempPtr→next = NULL;
delete tempPtr;

06-4 以指標實作環狀佇列

1、enqueue: newPtr→next = bockPtr→next
bockPtr→next = nextPtr

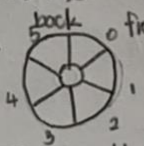


2. dequeue: QueueNode *tempPtr = bockPtr→next
if(bockPtr == bockPtr→next)
bockPtr = NULL;
else bockPtr→next = tempPtr→next;
tempPtr→next = NULL;
delete tempPtr;



stack

06-5 陣列實作佇列

1.


① Declare MaxQueue+1 for array item, see only MaxQueue for queue item
② Use a flag isFull to distinguish between full and empty

2. ADT
① statically — fixed-size queue can get full
prevents the enqueue opertation adding a item to queue, if array is full
② dynamically or pointer — no size restriction on the queue
③ point-based more efficient
ADT list is simpler to write, save programming time

06-7 事件驅動模擬

1. simulated time advance to the time of next event
using mathematical modle base on statistics and probability
2. Arrival events, Departure events
3. keeps track of arrival and departure events that will occur but have not occurred yet
Contain at most on arrival event and one departure event

06-9 多佇列模擬

1. single teller / single queues
Multiple teller / single queues
Multiple tellers / Multiple queues

summary: 1. FIFO, first in first out
2. circular array → problem of rightward drift
3. time-driven simulation, event-driven simulation
To implement on event-driven simulation, you maintain an event list that contains events that have not yet occurred

07-01 演算法的基本概念

1. Time efficiency, space efficiency
2. Specific implementation, computer, data
3. ex: Traverse a linked list of n nodes

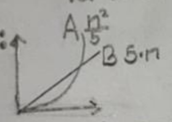
```
Node *cur = head;            1 assignments
while (cur != NULL){         n+1 comparisons
  cout << cur->item << endl; n write
  cur = cur->next;           n assignments
}
```

4. Algorithm's execution → number of operations
   ex: Towers of Hanoi → 2^n-1 moves

07-02 大O表示法

1. ex:
```
for(a=1; a<=n; a++)              n
  for(b=1; b<=a; b++)            1+2+…+n   n(n+1)/2
    for(c=1; c<=5; c++)          5
```

2. ex:



A: $O(n^2)$ B: $O(n)$

07-03 演算法複雜度分析

1. Definition of the order of an algorithm
   ⇒ if contants k and n_0 exist such that A requires no more than $k*f(n)$ time units to slove a problem of size $n \geq n_0$
   Algorithm A is order $f(n)$ – denoted $O(f(n))$

2. ex: $25n^2-25n$, $k=?$ $n_0=?$
   $k=3$, $n_0=n^2$
   ex: $(n+1)*(c+a)+n*w$, $k=?$ $n_0=?$
   k, $n_0=n$
   ex: 2^n-1, $k=?$ $n_0=?$
   $k=1$, $n_0=2^n$

07-04 複雜度成長函數

1、$O(n^3+3n)$ is $O(n^3)$ ⇒ ignore low-order terms

$O(5f(n)) = O(f(n))$ ⇒ ignore multiplicative constant in high-order term

$O(f(n)) + O(g(n)) = O(f(n) + g(n))$

2、$O(1) < O(\log_2 n) < O(n) < O(n * \log_2 n) < O(n^2) < O(n^3) < O(2^n)$

07-05 以搜尋為例說明演算法效率

1、frequently, seldom-used but critical operations must be efficient

2、Worst-case analysis, Average-case analysis, Best-case analysis

3、ex: sequential search

worst case: $O(n)$

Average case: $O(n)$

Best case: $O(1)$

ex: Binary search of a sorted array

worst case: $O(\log_2 n)$

Average case: $O(\log_2 n)$

Best case: $O(1)$

07-06 循序搜尋的效率比較

1、

	sorted	unsorted	found
Worse case	$O(n)$	$O(n)$	$O(n)$
Average case	$O(n)$	$O(n)$	$O(n)$
Best case	$O(1)$	$O(n)$	$O(1)$

07-07 排序演算法的分析

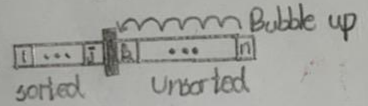
1、Internal sort
⇒ Requires that the collection of the data fit entirely in the computer's main memory

External sort
⇒ The collection of data will not fit in the computer's main memory all at once, but must reside in secondary storage

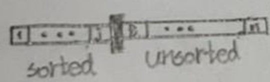
2、Stable sort vs. Unstable

Stable	Unstable
bubble	quick
insertion	heap
merge	
radix	

07-08 氣泡排序法

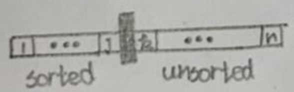
1、

07-09 選擇排序法

1、

Swap the smallest among ith to nth numbers with ith number

07-10 插入排序法

1、

Insert first element of unsorted section at the appropriate place in sorted section

07-11 簡易排序方法比較

1、

bubble sort	stable
selection sort	unstable
insertion sort	stable

07-12 氣泡排序法的複雜度分析

1、compare adjacent elements and exchange them if they are out of order
   Move the largest (smallest) to the end of the array
   Repeating this process eventually sorts the array into ascending (descending) order

2. worst case $O(n)$
   Best case $O(n)$

07-13 選擇排序法的複雜度分析

1、Place the largest (smallest) item in correct place
   Place the next largest (smallest) item in its correct palace, and so on

2、unstable
   worst case $O(n^2)$
   Best case $O(n^2)$

07-14 插入排序法的複雜度分析

1、Take the first item in the unsorted resorted region and place it into its correct position in the sorted region
   At each step, the sorted region grows by 1 and the unstored shrinks by 1

2、worst case (n^2)
   best case (n)

07-15 希爾排序

1、best case $O(n)$
   worst case $O(n\log^2 n)$

07-16,17,18 合併演算法

1. a recursive sorting algorithm
Devide an array into halves
sort each half
Merge the sorted halves into one sorted array
Devide-and-conquer

2. worst case: $O(n\log_2 n)$
Average case: $O(n\log_2 n)$

3. advantage → fast algorithm
disadvantage → require second array as large as the original array

07-19,20 快速演算法

1. divide-and-conquer algorithm
choose a pivot
partition the array about the pivot
items < pivot
items >= pivot
pivot is now in correct sorted position
sort the left section
sort the right section

2. Average case: $O(n \cdot \log_2 n)$
Worst case: $O(n^2)$
Even if the worst case occurs, quicksort's performance is acceptable for moderately large arrays

07-21,22,23 基數演算法

1. 10 is the radix of the decimal system
Treats a key as a character string
Repeatedly assign the keys into group (buckets) according to the ith character

2. worst case(n)
best case(n)

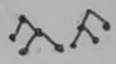
summary: 1. quicksort · mergesort are recursive algorithms
2. selection sort, bubble sort, insertion sort are $O(n^2)$ algorithms

Ch8 樹 tree

1. Terminology — ① composed of node and edges
② Parent-child ⇒ two nodes
Ancestor-descendant ⇒ among nodes
③ subtree — Any node and its descendant
④ general tree — one or more node
A single node r, the root
called subtrees of r
⑤ parent, child, root ⇒ only in the tre with no parent
subtree of node B ⇒ consists of a child of node B and the child descendants
Leaf ⇒ no children, sibling ⇒ common parent
Ancestor node B ⇒ on the path from root to B
Descendant node B ⇒ path from B to leaf

2. Binary tree — ① single node r, the root
② left subtree of r, right subtree of r

General Tree vs Binary Tree


or


3. Height of Tree — ① number of nodes along the longest path from root to leaf
② T is empty, height is 0
③ T is not empty height(T) = 1 + max{height(T_L), height(T_R)}

4. Full binary tree — ① node at levels < h have two children each
② T is empty, T is full binary tree height 0
T is not empty, root subtree both full binary tree of height h-1

5. complete binary tree — ① full to level h-1
② level is filled from left to right
③ level <= h-2, each two
level h-1, left

6. balanced binary tree — ① If the heights of any node's two subtree differ by no more than 1

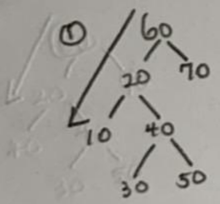
Full complete Balanced

7. Representations of Binary tree — ① use an array of tree nodes
② require creation of free list, keep track of available node
③ pointer-based representation
⇒ two pointers of nodes

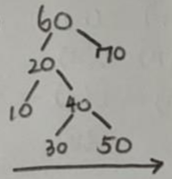
8. Traversals of a binary tree — ① Preorder traversal
visit root before visiting its subtrees
Before the recursive calls

② Inorder traversal
visit root between visiting its subtrees
Between the recursive calls

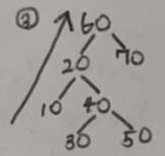
③ Postorder traversal
visit root after visiting its subtrees
After the recursive calls

① 

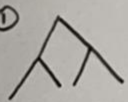
60, 20, 10, 40
30, 50, 70
preorder

② 

10, 20, 30, 40,
50, 60, 70
inorder

③ 

10, 30, 50, 40, 20, 70
60
postorder

① 
Full binary tree

② 
complete binary tree

9. Binary search tree — ① has to following properties for each node n

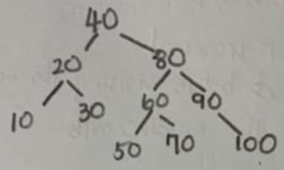
n's value is > all values in n's left subtree T_L

n's value is < all values in n's right subtree T_R

Both T_L, T_R are binary search tree

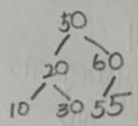
Insert:

ex: 40, 20, 10, 80, 90, 100, 30, 60, 50, 60



Delete:

ex: +55, -90, -70, -100, -40, -80



10. Efficiency — ① Retrieval $O(\log n)$ $O(n)$

Insertion $O(\log n)$ $O(n)$

Deletion $O(\log n)$ $O(n)$

Traversal $O(n)$ $O(n)$

② tree sort

Average $O(n\log n)$

worst $O(n^2)$