CS217 - Data Structures & Algorithm Analysis (DSAA)

Lecture #8

Elementary Data Structures

Prof. Pietro S. Oliveto
Department of Computer Science and Engineering
Southern University of Science and Technology (SUSTech)

olivetop@sustech.edu.cn
https://faculty.sustech.edu.cn/olivetop

Reading: Part III Introduction & Chapter 10

Aims of this lecture

- To introduce **data structures** and their typical operations.
- **Stacks**, **queues**, **priority queues** and **linked lists**.
- To work out the **running time** for operations on these data structures.
- To identify pros and cons for data structures in terms of efficiency.

Data Structures

- **Dynamic sets** that can store and retrieve elements.
- Data structures are techniques for representing finite dynamic sets of elements
- Each element can contain:
  - a **key**, used to identify the element
  - **Satellite data,** carried around but unused by the data structure
  - **Attributes,** that are manipulated by the data structure eg., pointers to other objects
- Often keys stem from a **totally ordered set** (e. g. numbers)
  - Allows to define the minimum, successor and predecessor

Data Structure Operations

- Operations on a dynamic sets S can be grouped into **queries** and **modifying operations**:
- Typical operations:
  - **Search(S, k):** returns a pointer to element $x \in S$ with **key k**, $(x.key = k)$ or NIL
  - **Insert(S, x):** adds element pointed to by x to S
  - **Delete(S, x):** given a pointer x to an element in S removes it from S
  - **Minimum(S), Maximum(S):** return a pointer x to element resp. with smallest or largest key
  - **Successor(S, x), Predecessor(S, x):** next larger (smaller) than $x.key$
- **Time** often measured using n as the number of elements in S.

Data Structure Operations

- What's the runtime of each operation on an **array**?
  - **Search(S, k):** returns a pointer to element $x \in S$ with **key k**, $(x.key = k)$ or NIL — $\Theta(n)$
  - **Insert(S, x):** adds element pointed to by x to S — $\Theta(1)$
  - **Delete(S, x):** given a pointer x to an element in S removes it from S — $\Theta(1)$
  - **Minimum(S), Maximum(S):** return a pointer x to element resp. with smallest or largest key — $\Theta(n)$
  - **Successor(S, x), Predecessor(S, x):** next larger (smaller) than $x.key$ — $\Theta(n)$

Data Structure Operations

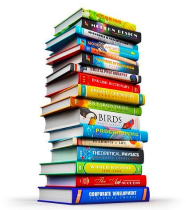
- What's the runtime of each operation on a **sorted array**?
  - **Search(S, k):** returns a pointer to element $x \in S$ with **key k**, $(x.key = k)$ or NIL — $\Theta(\log n)$
  - **Insert(S, x):** adds element pointed to by x to S — $\Theta(n)$
  - **Delete(S, x):** given a pointer x to an element in S removes it from S — $\Theta(n)$
  - **Minimum(S), Maximum(S):** return a pointer x to element resp. with smallest or largest key — $\Theta(1)$
  - **Successor(S, x), Predecessor(S, x):** next larger (smaller) than $x.key$ — $\Theta(1)$

We'll now see some data structures that improve on the array implementation for many of the dynamic-set operations.

Roadmap for the next lectures

- Simple data structures
  - Stacks
  - Queues
  - Linked lists
  - Binary search trees
  - Graphs
- Advanced data structures
  - Balanced trees
  - Priority queues

Stacks



3
6
8

- Only the **top element** is accessible in a stack.
  - Last-in, first-out policy (LIFO)
- Insert is usually called **Push**, and Delete is called **Pop**.

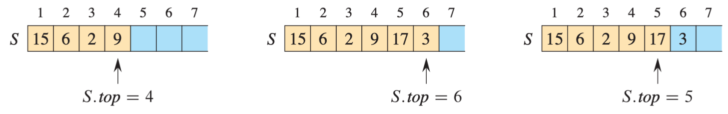
Stacks implemented using arrays

- Stacks can be implemented as an array S with attribute S.top.

```
PUSH(S, x)
  if S.top == S.size
      error "overflow"
  else S.top = S.top + 1
      S[S.top] = x
```

```
STACK-EMPTY(S)
  if S.top == 0
      return TRUE
  else return FALSE
```

```
POP(S)
  if STACK-EMPTY(S)
      error "underflow"
  else S.top = S.top - 1
      return S[S.top + 1]
```



(a) (b) (c)

- All stack operations take time $O(1)$.

Stacks Application (1): Bracket Balance Checking

- $1 + \{2 * [x + (4y - z)] * [5x - (5y + z)] - 5t\}$
- {[()][()]}
- Are the brackets correctly balanced or not?
- Read the expression: **Push** each opening bracket and **pop** for each closing bracket
- If the type of popped bracket always matches return **true,** else return **false**
- What's the runtime of the algorithm? \Omega(n)

Stacks Application (2): Postfix expression

- $5 * ((9 + 3) * (4 * 2) + 7)$ (infix expression)
- 5 9 3 + 4 2 * * 7 + * (postfix expression)
- Parsing postfix expressions is somewhat easier than infix expressions. Why?
- Read the tokens one at a time:
  - If it is an operand, **push** it on the stack
  - If it is a binary operator **pop** twice, apply the operator, and **push** the result back on the stack
- What is the runtime of the algorithm?

Stacks Application (2): Postfix expression

- $5 * ((9 + 3) * (4 * 2) + 7)$ (infix expression)
- 5 9 3 + 4 2 * * 7 + * (postfix expression)

Stack operations	Stack elements
push(5)	5
push(9)	5 9
push(3)	5 9 3
push(pop() + pop())	5 12
push(4)	5 12 4
push(2)	5 12 4 2
push(pop() * pop())	5 12 8
push(pop() * pop())	5 96
push(7)	5 96 7
push(pop() + pop())	5 103
push(pop() * pop())	515

Queues

head	3	6	8	tail

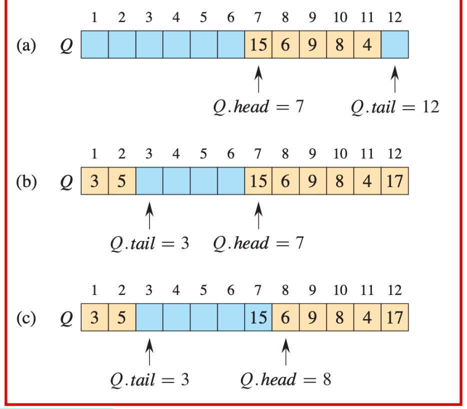
- The British love them ☺
- The first element in a queue is accessible.
  - First-in, first-out policy (FIFO)
- Insert is called **Enqueue**, Delete is called **Dequeue**.
- Queues have a **head** and a **tail**, like in a supermarket
  - Elements are added to the tail
  - Elements are extracted from the head

Queues implemented using arrays

- Queues can be stored in an array "**wrapped around**".

```
ENQUEUE(Q, x)
  Q[Q.tail] = x
  if Q.tail == Q.size
      Q.tail = 1
  else Q.tail = Q.tail + 1
```

```
DEQUEUE(Q)
  x = Q[Q.head]
  if Q.head == Q.size
      Q.head = 1
  else Q.head = Q.head + 1
  return x
```



- All queue operations take time $O(1)$.

当 `tail` "刚好在 `head` 的前一个"（即 `(tail + 1) % cap == head`）时，确实还有**一个物理空格**，就是 `tail` 自己那个格子。但这个格子是**专门保留的哨兵**，用来区分"空"（`head==tail`）与"满"（`(tail+1)%cap==head`）——我们**刻意不使用它**。因此这时称为"满"，意思是**逻辑上不能再入队**，不是说数组里一个空格都没有。

Queues: Applications

- Playlists (eg., iTunes)
- Dispensing requests on a shared resource (eg., a printer, a server, a processor etc.,)
- Data buffers (eg., streaming services)
- What if I have priorities on the use of the resource?

Priority Queues: Motivation

- Schedule jobs on a computer shared among multiple users
- A max-priority queue keeps track of the jobs to be performed and their relative priorities
- When a job is finished the scheduler selects the job with highest priority from those pending
- Jobs can be added to the scheduler at any time

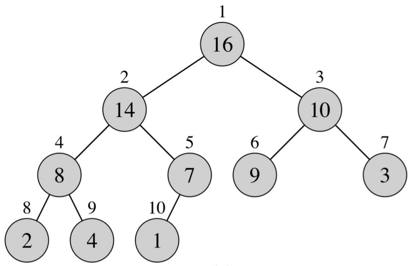
Job	Owner	Priority (key)
Job 1	Tang Ke	35
Job 12	Oliveto Pietro	2
Job 24	Hao Qi	22
Job 25	Yu Shiqi	18
Job 72	Tang Ke	30

- **Use a heap!**

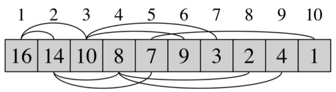
➢ Heap Properties

- **Max-heap property**: for every node other than the root, the parent is no smaller than the node, $A[Parent(i)] \geq A[i]$.

- In a max-heap, the **root** always stores a **largest** element.
  → this is what we want!



(a)



(b)

- **Min-heap property**: for every node other than the root, the parent is no larger than the node, $A[Parent(i)] \leq A[i]$.

➢ Priority Queue based on max-heap

- A data structure for maintaining a set S of elements with an associated element called key (the priority).

Operation	Time
Insert(S, x, k) – inserts *x* with key *k* into *S*	$O(\log n)$
Maximum (S) – returns the element in *S* with the largest key	$O(1)$
Extract-Max(S) – removes and returns element in *S* with the largest key	$O(\log n)$
Increase-Key(S, x, k) – increases they key of *x* to a larger value *k* (element may float up in the heap)	$O(\log n)$

Min-priority queue based on min-heap also exist: we will use them in graph algorithms (eg., Djikstra, Prim)

➢ Priority Queues: handles

Operation	Time
Insert(S, x, k) – inserts *x* with key *k* into *S*	$O(\log n)$
Maximum (S) – returns the element in *S* with the largest key	$O(1)$
Extract-Max(S) – removes and returns element in *S* with the largest key	$O(\log n)$
Increase-Key(S, x, k) – increases they key of *x* to a larger value *k* (element may float up in the heap)	$O(\log n)$

- Elements in the priority queue correspond to objects (eg., jobs)
- For an operation such as Increase-Key we need a way to **map** objects to and from their position in the heap (and update it as it moves in the heap)
- One way is to use **handles:** extra information stored in the object that allows to do the mapping
- A Job x: **x.key** (priority) **x.heap_index** (handle) x.satellite_data;
- The heap needs to contain for each element **a pointer to the object**

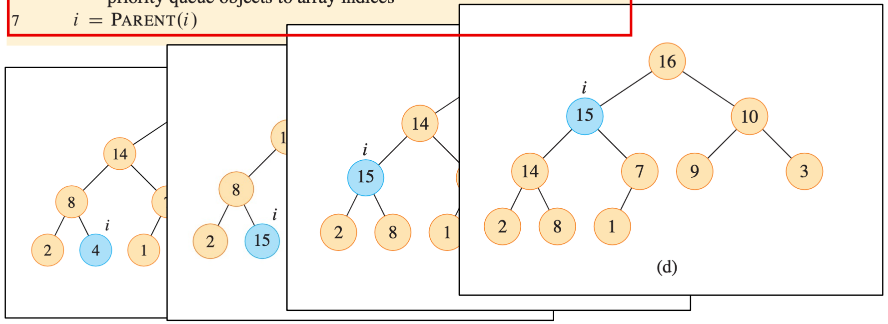
➢ Find and extract next job

```
MAX-HEAP-MAXIMUM(A)
  if A.heap-size < 1
      error "heap underflow"
  return A[1]
```

```
MAX-HEAP-EXTRACT-MAX(A)
  max = MAX-HEAP-MAXIMUM(A)
  A[1] = A[A.heap-size]
  A.heap-size = A.heap-size - 1
  MAX-HEAPIFY(A, 1)
  return max
```

➢ Increase job priority

```
MAX-HEAP-INCREASE-KEY(A, x, k)
if k < x.key
    error "new key is smaller than current key"
x.key = k
find the index i in array A where object x occurs      i = x.heap_index()
while i > 1 and A[PARENT(i)].key < A[i].key
    exchange A[i] with A[PARENT(i)], updating the information that maps
        priority queue objects to array indices
    i = PARENT(i)
```



➢ Insert new job

```
MAX-HEAP-INSERT(A, x, n)
if A.heap-size == n
    error "heap overflow"
A.heap-size = A.heap-size + 1
k = x.key
x.key = -∞
A[A.heap-size] = x
map x to index heap-size in the array
MAX-HEAP-INCREASE-KEY(A, x, k)
```

把Max_Heapify的工作交给Increase去做，对heap进行repair

▶ Linked Lists: Array Disadvantages

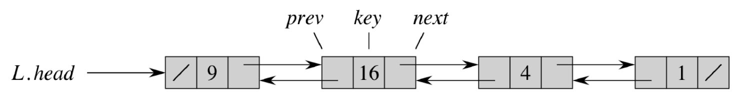
- You need to specify an initial size
- Changing the size of an array is troublesome
- Inserting and deleting elements in specific positions is difficult
- Let's say we want to delete 10 and keep the order of the rest:

A	5	8	10	13	16	19	27	46	51	86
A	5	8		13	16	19	27	46	51	86
A	5	8	13	16	19	27	46	51	86	

- What's the time complexity?

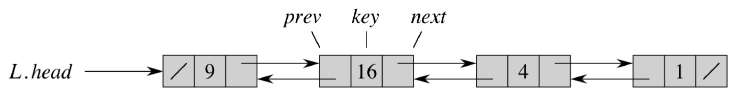
▶ Linked Lists

- Objects are linked using **pointers to the next element.**
- Linked lists can be **singly linked** or **doubly linked**: pointers to next and previous elements.
- Each element x has attributes
  - x.key – the key used to identify the element
  - x.next – a pointer to the next element
  - x.prev – a pointer to the previous element
  - Optional: further satellite data



KEY: We don't know any element's position/index, we only have its next and previous information, which combine all elements into a linked list.

Linked Lists: Searching



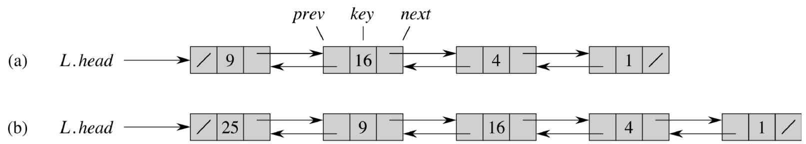
- Search inspects all elements in sequence and stops when the key has been found or the end of the list is reached.

```
LIST-SEARCH(L, k)
x = L.head
while x ≠ NIL and x.key ≠ k do
    x = x.next
return x
```

- The worst-case time is $\Theta(n)$, since it may have to search the entire list.

Here, we only deal with the 'head' pointer. So we insert things to the front. If we deal with the 'tail' pointer, we can also insert things to the endof the list,.

Linked Lists: Inserting at the front



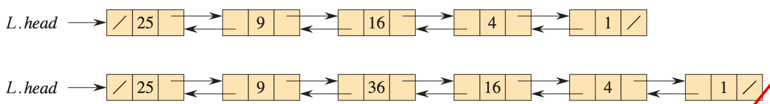
- New elements are added to the front of the list.

```
LIST-PREPEND(L, x)
x.next = L.head
x.prev = NIL
if L.head ≠ NIL
    L.head.prev = x
L.head = x
```

- The time for an insertion is $O(1)$.

add和insert是相互补全配合的！

Linked Lists: Inserting after element y

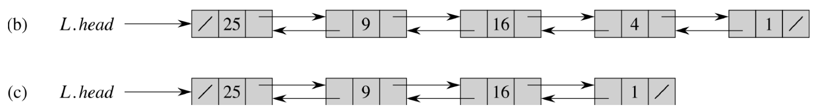


- New element added after element y.

```
LIST-INSERT(x, y)
x.next = y.next
x.prev = y
if y.next ≠ NIL
    y.next.prev = x
y.next = x
```

- The time for an insertion is $O(1)$ if you know the pointer to y

Linked Lists: Deleting



- If element x is known, update pointers to take it out.

```
LIST-DELETE(L, x)
if x.prev ≠ NIL then
    x.prev.next = x.next
else
    L.head = x.next
if x.next ≠ NIL then
    x.next.prev = x.prev
```

- The time for a deletion is $O(1)$.
  But if we only have the key and need to search the element x, it's time $\Theta(n)$ in the worst case.

Summary

- **Stacks** and **Queues** are simple data structures that can
  - be implemented efficiently in arrays (modulo space issues)
  - Have a restricted set of operations, but these run in time $O(1)$.
- **Priority Queues**: all operations in at most $O(\log n)$ time
- Linked lists form an **unordered list** of elements
  - **Insertion** is fast if not important where it occurs: time $O(1)$.
  - **Searching** takes worst-case time $\Theta(n)$.
  - **Deletion** runs in time $O(1)$ if the element is known, otherwise we need to run a search beforehand and incur time $\Theta(n)$.