بسم الله الرحمن الرحیم

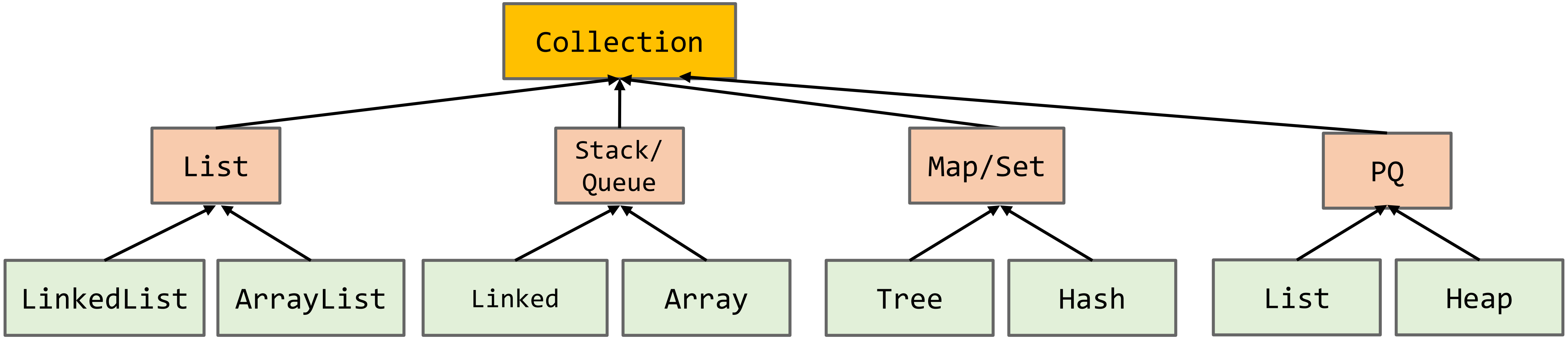
ساختمان‌های داده

جلسه ۲۴

مجتبی خلیلی
دانشکده برق و کامپیوتر
دانشگاه صنعتی اصفهان

ساختارهای داده

IUT-ECE



Priority Queues

Introduction

- Priority Queue
  - Data structure for storing a collection of prioritized elements
  - Supporting arbitrary element insertion
  - Supporting removal of elements in order of priority

- So far, we covered "position-based" data structures
  - Stacks, queues, deques, lists, and even lists
  - Store elements at specific positions (linear or hierarchical)
  - Insertion and removal based on "position" (linear or hierarchical)
  - But, priority queue
    - Insertion and removal: priority-based

- Question: how to express the priority of an element
  - Key (example: your student id)

Priority Queue ADT

- A priority queue stores a collection of entries
- Typically, an entry is a pair (key, value), where the key indicates the priority
- Main methods of the Priority Queue ADT
  - insert(e)
    inserts an entry e (with an implicit associated key value)
  - removeMin()
    removes the entry with smallest key
- Additional methods
  - min()
    returns, but does not remove, an entry with smallest key
  - size(), empty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

CLRS definition

A ***priority queue*** is a data structure for maintaining a set S of elements, each with an associated value called a ***key***. A ***max-priority queue*** supports the following operations:

INSERT(S, x, k) inserts the element x with key k into the set S, which is equivalent to the operation $S = S \cup \{x\}$.

MAXIMUM(S) returns the element of S with the largest key.

EXTRACT-MAX(S) removes and returns the element of S with the largest key.

CLRS definition

Alternatively, a ***min-priority queue*** supports the operations INSERT, MINIMUM, EXTRACT-MIN,

Total Order Relations (a topic of Discrete Math)

- Keys in a priority queue can be arbitrary objects on which an order is defined
- Two distinct entries in a priority queue can have the same key
- Total ordering
  - Comparison rule should be defined for every pair of keys
- Mathematical concept of total order relation $\leq$
  - Reflexive property: $x \leq x$
  - Antisymmetric property: $x \leq y \wedge y \leq x \Rightarrow x = y$
  - Transitive property: $x \leq y \wedge y \leq z \Rightarrow x \leq z$
- Satisfying the above three properties ensures:
  - Never leading to a comparison contradiction

Example: Total order & Partial order

- 2D points with (x-coordinate, y-coordinate)
  - Define relation '>=' based on x-first, and y-next
  - (4,3) >= (3,4), (3,5) >= (3,4)
  - Total ordering

  - What about defining relation '>=' based on both x and y
  - (4,3) >=(2,1), but (4,3) ??? (3,4)
  - Partial ordering
    - Comparison not defined for some objects

- We assume that we define a comparison that leads to total ordering.

Comparator

- کلیدها و مقادیر
- Composition method (k,e)
- Comparator

Comparator

- به تعریف یک مقایسه کننده نیاز داریم (تابع C).
- وقتی int ها را مقایسه میکنیم C بسیار ساده است: =>

Comparator(How to define order for any object?)

- 2D points with (x-coordinate, y-coordinate)

- How to design “comparison logic” in a programming language?

- What design is good?

Design 1: Separate Design

- Different Priority Queue based on the element type and the manner of comparing elements
- PQ_Int, PQ_2D, PQ_XXX
- Simple, but not general
- Many copies of the same code

Design 2: Template and Overloading


```
bool operator<(const Point2D& p, const Point2D& q) {
  if (p.getX() == q.getX())    return p.getY() < q.getY();
  else                         return p.getX() < q.getX();
}
```

- General enough for many situations
- But,
  - Cannot have multiple comparison methods for the same type
  - What about comparison based on y-first, and x-next?

- Even for the same data type, we want to apply different comparison methods A or B, depending on the situations

Design 3: Separating Comparator

- 2D points:
  - Sometimes we want either of
    X-based comparison, Y-based comparison
- Idea
  - Define a comparator class, e.g., “LeftRight” (x-based) and “BottomTop” (y-based)
  - Overload “()” operator

```
class LeftRight {                                    // a left-right comparator
public:
  bool operator()(const Point2D& p, const Point2D& q) const
    { return p.getX() < q.getX(); }
};

class BottomTop {                                    // a bottom-top comparator
public:
  bool operator()(const Point2D& p, const Point2D& q) const
    { return p.getY() < q.getY(); }
};
```

Design 3: Separating Comparator

IUT-ECE

```
Point2D p(1.3, 5.7), q(2.5, 0.6);     // two points
LeftRight leftRight;                  // a left-right comparator
BottomTop bottomTop;                  // a bottom-top comparator
printSmaller(p, q, leftRight);        // outputs: (1.3, 5.7)
printSmaller(p, q, bottomTop);        // outputs: (2.5, 0.6)
```

```
template <typename E, typename C>     // element type and comparator
void printSmaller(const E& p, const E& q, const C& isLess) {
  cout << (isLess(p, q) ? p : q) << endl;  // print the smaller of p and q
}
```

In C++

IUT-ECE

```
#include <algorithm>
#include <functional>
#include <array>
#include <iostream>

// sort using a custom function object
struct MyLess{
  bool operator()(int a, int b) const
  {        return a > b;  }
};


int main()
{
    std::array<int, 10> s = {5, 7, 4, 2, 8, 6, 1, 9, 0, 3};

    // sort using the default operator<
    std::sort(s.begin(), s.end());
    for (int i=0 ; i<s.size();i++) {
        std::cout << s[i] << " ";
    }
    std::cout << '\n';

    MyLess myless;

    std::sort(s.begin(), s.end(), myless);

    for (int i=0 ; i<s.size();i++) {
        std::cout << s[i] << " ";
    }
    std::cout << '\n';

}
```

```
0 1 2 3 4 5 6 7 8 9
9 8 7 6 5 4 3 2 1 0
```

Priority Queue ADT

- A priority queue stores a collection of entries
- Typically, an entry is a pair (key, value), where the key indicates the priority
- Main methods of the Priority Queue ADT
  - insert(e)
    inserts an entry e (with an implicit associated key value)
  - removeMin()
    removes the entry with smallest key
- Additional methods
  - min()
    returns, but does not remove, an entry with smallest key
  - size(), empty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

Priority Queue (example)

IUT-ECE

Operation	*Output*	*Priority Queue*
insert(5)	–	{5}
insert(9)	–	{5,9}
insert(2)	–	{2,5,9}
insert(7)	–	{2,5,7,9}
min()	[2]	{2,5,7,9}
removeMin()	–	{5,7,9}
size()	3	{5,7,9}
min()	[5]	{5,7,9}
removeMin()	–	{7,9}
removeMin()	–	{9}
removeMin()	–	{}
empty()	*true*	{}
removeMin()	*“error”*	{}

Priority Queue Sorting

- We can use a priority queue to sort a set of comparable elements
  1. Insert the elements one by one with a series of insert operations
  2. Remove the elements in sorted order with a series of removeMin operations

- The running time of this sorting method depends on the priority queue implementation

Priority Queue Sorting

IUT-ECE

Algorithm PriorityQueueSort(L, P):

Input: An STL list L of n elements and a priority queue, P, that compares elements using a total order relation

Output: The sorted list L

while $!L.\mathsf{empty}()$ **do**
  $e \leftarrow L.\mathsf{front}$
  $L.\mathsf{pop\_front}()$ {remove an element e from the list}
  $P.\mathsf{insert}(e)$ {... and it to the priority queue}
while $!P.\mathsf{empty}()$ **do**
  $e \leftarrow P.\mathsf{min}()$
  $P.\mathsf{removeMin}()$ {remove the smallest element e from the queue}
  $L.\mathsf{push\_back}(e)$ {... and append it to the back of L}

List-based Priority Queue

- Implementation with an unsorted list

- Performance:
  - insert takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - removeMin and min take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

- Implementation with a sorted list

- Performance:
  - insert takes $O(n)$ time since we have to find the place where to insert the item
  - removeMin and min take $O(1)$ time, since the smallest key is at the beginning

List-based Priority Queue

DS	Insert	Remove min
Unsorted Array	O(1)	O(N)
Unsorted Linked List	O(1)	O(N)
Sorted Array	O(N)	O(1)
Sorted Linked List	O(N)	O(1)
?	?	?

Insertion-Sort

- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted List
- Running time of Insertion-sort:
  1. Inserting the elements into the priority queue with n insert operations takes time proportional to
     $$1 + 2 + \ldots + n$$
  2. Removing the elements in sorted order from the priority queue with a series of n removeMin operations takes $O(n)$ time
- Insertion-sort runs in $O(n^2)$ time

Insertion-Sort Example

	Sequence/List S	Priority queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(4,7)
(c)	(2,5,3,9)	(4,7,8)
(d)	(5,3,9)	(2,4,7,8)
(e)	(3,9)	(2,4,5,7,8)
(f)	(9)	(2,3,4,5,7,8)
(g)	()	(2,3,4,5,7,8,9)
Phase 2		
(a)	(2)	(3,4,5,7,8,9)
(b)	(2,3)	(4,5,7,8,9)
..	..	..
(g)	(2,3,4,5,7,8,9)	()

Selection-Sort

- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted list
- Running time of Selection-sort:
  1. Inserting the elements into the priority queue with $\boldsymbol{n}$ insert operations takes $\boldsymbol{O}(\boldsymbol{n})$ time
  2. Removing the elements in sorted order from the priority queue with $\boldsymbol{n}$ removeMin operations takes time proportional to

$$1 + 2 + \ldots + \boldsymbol{n}$$

- Selection-sort runs in $\boldsymbol{O}(\boldsymbol{n}^2)$ time

Selection-Sort Example

	Sequence/List S	Priority Queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(7,4)
..	..	..
(g)	()	(7,4,8,2,5,3,9)
Phase 2		
(a)	(2)	(7,4,8,5,3,9)
(b)	(2,3)	(7,4,8,5,9)
(c)	(2,3,4)	(7,8,5,9)
(d)	(2,3,4,5)	(7,8,9)
(e)	(2,3,4,5,7)	(8,9)
(f)	(2,3,4,5,7,8)	(9)
(g)	(2,3,4,5,7,8,9)	()