DA-5

Lorenz Pfäffli 18-925-230
Lukas Ingold 20-123-998

Datenstrukturen und Algorithmen
Übung 5 – Elementare Datenstrukturen

Abgabefrist: 01.04.2021, 16:00 h

Verspätete Abgaben werden nicht bewertet.

Theoretische Aufgaben

1. Geben Sie eine in Zeit $\Theta(n)$ laufende, nichtrekursive Prozedur an, welche die Reihenfolge einer **einfach** verketteten Liste aus n Elementen umkehrt. Die Prozedur sollte nur konstant viel Speicherplatz benutzen (abgesehen vom Speicherplatz, der für die Liste selbst gebraucht wird). **(1 Punkt)**

2. Schreiben Sie Pseudocode, um eine Warteschlange mit einer einfach verketteten Liste zu implementieren. Ihre Lösung soll Code für die Operationen ENQUEUE und DEQUEUE enthalten. Nehmen Sie an, die Listenelemente hätten ein Feld *next* mit dem Zeiger auf das nächste Element, und ein Feld *key* mit dem Schlüssel. Die Operationen ENQUEUE und DEQUEUE sollten noch immer in Zeit $\mathcal{O}(1)$ arbeiten.

   Illustrieren Sie den Ablauf der folgenden Operationen, indem Sie für jeden Schritt die Liste darstellen und gegebenenfalls den Rückgabewert angeben: *ENQUEUE(3); ENQUEUE(5); DEQUEUE(); ENQUEUE(2); DEQUEUE(); ENQUEUE(8); ENQUEUE(9); DEQUEUE(); DEQUEUE(); DEQUEUE(); DEQUEUE()* **(1 Punkt)**

3. Schreiben Sie Pseudocode für eine rekursive Prozedur, die alle Knoten eines gerichteten Baumes mit unbeschränktem Grad besucht und jeweils den Schlüssel des Knotens ausgibt. Nehmen Sie an, die Knoten des Baumes hätten folgende Felder: *key* für den Schlüssel, *left-child* für den Zeiger auf das sich am weitesten links befindende Kind und *right-sibling* für den Zeiger auf das rechte Geschwister. **(1 Punkt)**

4. Schreiben Sie Pseudocode für eine *nicht-rekursive Prozedur*, die alle Knoten eines gerichteten Baumes mit unbeschränktem Grad besucht und jeweils den Schlüssel des Knotens ausgibt. Verwenden Sie dazu einen Stack. Nehmen Sie an, der Stack unterstützt die Operationen PUSH(Node) und POP, wobei *node* ein Knoten des Baumes ist. Der Rückgabewert von POP ist ein Knoten *node* oder NIL wenn der Stack leer ist. **(1 Punkt)**

5. Geben Sie Pseudocode für eine Methode MERGE an, die zwei *sortierte* einfach verkettete zyklische Listen als Parameter annimmt und diese in linearer Zeit zu einer einzelnen *sortierten* Liste zusammenfügt. Die ursprünglichen Listen dürfen dabei zerstört werden und es soll nur konstant viel zusätzlicher Speicher verwendet werden.

   Wieso ist die Zeitkomplexität quadratisch statt linear, wenn der MERGE Pseudocode aus Kapitel 2, Seite 32 im Buch direkt verwendet wird, die Felder A, L, R aber durch verkettete Listen ersetzt werden? **(1 Punkt)**

Praktische Aufgaben

In dieser Aufgabe werden Sie einen k-d-Tree implementieren. Wir stellen auf Ilias Code zur Verfügung, auf dem Sie aufbauen können. Der Code enthält eine Klasse `KDTreeTester`, welche das

Programm startet und ein Fenster mit zufällig generierten Punkten anzeigt. Die Variabeln w, h und n steuern die Grösse des Fensters und die Anzahl Punkte.

Die Klasse *KDTreeVisualization* enthält verschiedene Funktionen zum Generieren und Anzeigen der zufälligen Punkte. Die Punkte werden in einer verketteten Liste gespeichert. Die Klasse enthält auch eine Funktion um die Reihenfolge der Punkte in der verketteten Liste zu visualisieren.

Eine detaillierte Beschreibung von k-d-Bäumen finden Sie auf Wikipedia.

1. Schreiben Sie eine Funktion, die für einen gegebenen Punkt seinen nächsten Nachbarn in der Liste sucht. Implementieren Sie dazu die Funktion `listSearchNN` in `KDTreeVisualization`. Sie können Ihre Funktion testen, indem Sie im Menu *Search "Search List for NN"* wählen. Die aufgerufene Funktion sucht den nächsten Nachbarn für x Punkte und misst dabei die benötigte Zeit. **(1 Punkt)**

2. Implementieren Sie die Funktion `createKDTree`, welche die Punkte aus der Liste in einem k-d-Baum speichert. Nutzen Sie dazu die innere Klasse `TreeNode`. Ein Objekt dieser Klasse repräsentiert einen Knoten im k-d-Baum. Die Variable `kdRoot` soll die Wurzel Ihres Baumes enthalten.

   Um die Punktelisten zu sortieren, können Sie die Klasse *PointComparator* und die Java Funktion `Collections.sort(List list, Comparator c)` verwenden.

   Sie können den k-d-Baum mit *Visualize kd-Tree* anzeigen lassen. **(2 Punkte)**

3. Schreiben Sie nun eine Funktion, welche die Suche nach dem nächsten Nachbarn auf dem k-d-Baum durchführt. Implementieren Sie dazu die Funktion `treeSearchNN`. Vergleichen Sie dann die Laufzeit der Suche auf dem k-d-Baum mit der Suche auf der Liste. Führen Sie dazu eine Suche nach dem nächsten Nachbarn für verschiedene Mengen zu suchender Punkte und mehrere unterschiedliche Punktemengen von unterschiedlicher Grösse durch. Stellen Sie die Resultate in einer Liste und grafisch dar. Stimmen Ihre Messungen mit der theoretisch erwarteten Zeitkomplexität überein (Suche in der Liste: $\mathcal{O}(n)$, Suche im Baum: $\mathcal{O}(\lg n)$)? **(2 Punkte)**

Vergessen Sie nicht Ihren Sourcecode innerhalb der Deadline über die Ilias Aufgabenseite einzureichen.

1.)

```
Listeumkehren ( Liste)

if   Liste. size()  ≤  1        Nur einmal
{
   return  Liste
}
else {
                                  Θ(n)
     for   i = 1   to   Liste.size ()
     {
        Liste [i].next   =  point to ( Liste [i-1])    // .next returns a pointer to the next field
     }                                                 // point to (List[i]) returns the adress of given Element

     return  List
   }
```

2.)

```
void  enquene  ( Element  element )
{
   element.next  =  null

   (tail.next).next  =  zeige Auf (element)    // Element vor tail zeigt auf element

   tail.next         =  zeige Auf (element)    // tail soll auf element zeigen
}

Element  dequene ()
{
  if (tail.next == head)       // wenn Liste leer wird null zurückgegeben
  {
    return  null
  }
  else
  {
    Element  k  =  head.next
    head.next   =  zeige Auf ((head.next). next)

    if ( head.next == null)
    {
      tail.next = zeige Auf (head)
    }
    return k
  }
```

=> { (head) ← (tail) }

ENQUEUE (3) : { (head) → 3 ← (tail) }

ENQUEUE (5): { (head) → 3 → 5 ← (tail) }

DEQUEUE () : { (head) → 5 ← (tail) }, return 3

ENQUEUE (2): { (head) → 5 → 2 ← (tail) }

DEQUEUE () : { (head) → 2 ← (tail) }, return 5

ENQUEUE (8): { (head) → 2 → 8 ← (tail) }

ENQUEUE (9): { (head) → 2 → 8 → 9 ← (tail) }

DEQUEUE () : { (head) → 8 → 9 ← (tail) }, return 2

DEQUEUE () : { (head) → 9 ← (tail) }, return 8

DEQUEUE () : { (head) ← (tail) }, return 9

DEQUEUE () : { (head) ← (tail) }, return null

3.)

```
getKey (node  n)
{
  print  n.key

  if ( n.left - child  != null)
  {
    getKey (n.left - child)
  }
  if ( n.right - sibling  != null)
  {
    getKey (n.right-sibling)
  }
}
```

4.)

```
getKey (node  n)
{
  while  stack  != empty
    {
        print  n

        if (n.left - child  != null)
        {
          stack.push (n.left - child)
        }
        if (n.right - sibling  != null)
        {
          stack.push (n.right - sibling)
        }
        n = stack. pop

    }

  }
```

A5)

```
// Precondition : Lists L, R are non empty lists

0  MERGE-LISTS ( List L , List R ) {
1      current = R.nil.next ;  current_prev = R.nil ;
2      while ( ! L.isEmty() ) {
3          element = L.nil.next                  } "pop" first element in L
4          L.nil.next = element.next             }
5          while ( current.next.key ≤ element.key AND current.next != R.nil ) {
6              current_prev = current
7              current = current.next
8          }
9          if ( current.key ≤ element.key ) {
10             current_next = current.next
11             current.next = element
12             element.next = current_next
13         } else {
14             current_prev.next = element
15             element.next = current
16         }
17     }
```

b) Die Zeitkomplexität der MERGE Funktion (s. 32) wäre quadratisch statt linear, da der Zugriff auf Elemente in der Liste nicht konstant ist.

Der Zugriff auf Element mit gegebenem Index kostet in Arrays $O(1)$.
Der Zugriff auf Element mit gegebenem Index kostet in einzeln verketteten Listen kostet $O(n)$.

In der Scheife (z. 12 - 17) würde man also für jeden Listenzugriff lineare Laufzeit benötigen.