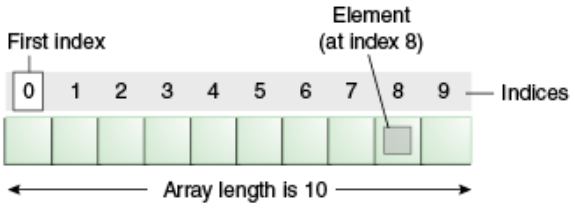
מבני נתונים

מערך



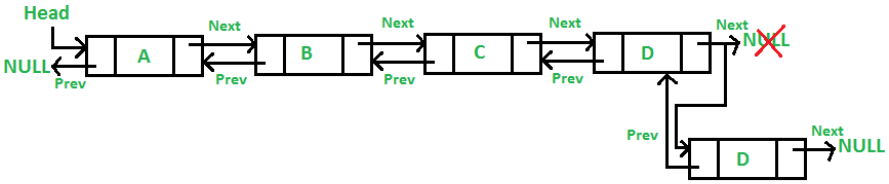
יתרונות:

- גישה מהירה לאיברים במערך
- הוצאה / הופסה מהירה (PUSH/POP)
- הכנסה לפי הסדר

חסרונות

- הכנסה איטית
- מחיקה איטית

רשימה מקושרת



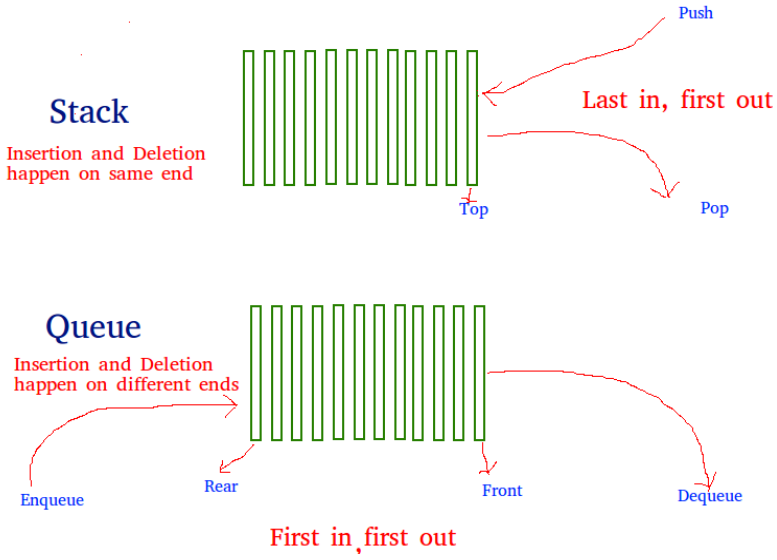
יתרונות

- הכנסה /מחיקרה מהירה
- בעל סדר מסוים
- גודל משתנה

חסרונות

- גישה אטית לאיברים
- תופס הרבה מקום בזיכרון

תור / מחסנית



יתרונות

- הכנסה /מחיקה מהירה
- גישה מהירה לאיבר הראשון
- בעל סדר מסוים
- גודל משתנה

חסרונות

- גישה אטית לאיברים

מיונים

מיון בועה

עובר על הרשימה
אם איבר ימין גדול מאיבר שמאל
מבצע החלפה

- לא שימושי
- סיבוכיות מקום O(1)
- סיבוכיות זמן O(n^2)

מיון בחירה

עובר על הרשימה
מוצא את המספר הקטן ביותר
ומעביר אותו להתחלה
עד המערך ממויון

- לא שימושי
- סיבוכיות מקום O (1)
- סיבוכיות זמן O(n^2)

מיון הכנסה

עובר על הרשימה
אם מוצא מספר קטן יותר מהמספר השמאלי
מבצע הסטה של הערך שמאלה

- יעיל ברישמות קצרות וכימעט ממוינות Ω(n)
- סיבוכיות מקום O(1)
- סיבוכיות זמן O(n^2)

מיון מיזוג

מפצל את הרשימה בצורה רקורסיבית עד שאי אפשר לפצל יותר
ממזג חזרה את הרשימות הקטנות שנצרו בצורה ממוינת
עד להחזרת המערך השלם הממוין

- אותה יעילות תמיד O(nlogn) = Ω(nlogn)
- טוב בקבצים גדולים
- סיבוכיות מקום O (n)
- סיבוכיות זמן O(nlogn)

מיון מהיר

בוחרים ערך pivot שמחלק את מערך לשניים
בצורה רקורסיבית בודקים את האיבר האחרון והראשון
אם האיבר משמאל קטן מהpivot מתקמדים לאיבר הבאה
אם האיבר הימני גדול מהpivot מתקמדים לאיבר הבאה
אם האיבר השמאלי גדול מהpivot והימני קטן מהpivot מחליפים בניהם

- סיבוכיות זמן טובה ביותר O (nlogn) יעיל כאשר המערך אינו ממונין ושיש מידע על הנתונים לבחירה של טובה של pivot
- סיבוכיות זמן גרועה ביותר O (n^2) גרוע שהמערך כימעט ממוין ואין מידע על ערך גבי ערך הpivot
- סיבוכיות זמן ממוצעת O(nlogn) עם המהירות ממוצעת הכי טובה כאשר הpivot הוא החציון
- סיבוכיות מקום O (nlogn)

מיון ספריה

יוצרים רשימה עם כל המספרים בטווח הנתון
עוברים על הרשימה וסופרים כל מספר כמה פעמים הוא חוזר
ומחזרים את המספרים לפי הסדר

- אינו מבוסס השוואה
- עובד רק על מספרים
- חייב לקבל טווח של מספרים מראש
- יעיל כאשר הטווח של המספרים במערך הוא קטן
- כדאי להשתמש כאשר יש טווח של מספרים ידוע והרבה מספרים חוזרים על עצמם
- סיבוכיות מקום O(n+k)
- סיבוכיות זמן O(k)

מיון בסיס

מיון בסיס עושה שימוש במיון ספירה כדי למיין מערכים של מספריסאו מחרוזות. האלגוריתם מפעיל את מיון ספירה D פעמים, כאשר D הוא כמות הספרות של המספר הגדול ביותר בקלט. זמן הריצה של מיון ספירה הוא כאמור N . O(N + K) מייצג את כמות הערכים אותם אנו נדרשים למיין, ו- K הוא הבסיס של המספרים הנתונים
(אם אנחנו ממיינים מספרים בבסיס 10, אז K = 10). לרוב הבסיס ידוע, ולכן K הוא זניח. כתוצאה מכך יש מי שמתאר את זמן הריצה של האלגוריתם כ- O (D * N) כאשר הסימונים עשויים באופן טבעי להשתנות ממקור למקור.
(לטענת אחרים גם D) שמייצג כאמור את כמות הספרות של המספר הגדול ביותר בקלט חסום תמיד על ידי איזשהו קבוע,ולכן הם מתארים את זמן הריצה של אלגוריתם מיון בסיס כ- O(N)

- אינו מבוסס השוואה
- ניתן למיין גם מספרים וגם מחרוזות
- טוב להשתמש במערך כאשר מספר הספרות בקלט קטן
- דורש טווח של ערכים
- סיבוכיות מקום O(d*n)
- סיבוכיות זמן O(k+n)

מימוש של רשימה מקושרת דו כיוונית

```
public DoublyLinkedList()
{
    head=null;
    tail=null;
    size=0;
}

public void addFirst(Node data)
{
    if (head == null && tail== null)
        head = tail = data;

    else {
        data.next=head;
        head.prev=data;
        head=data;}
    size++;
    }


public void addLast(Node data)
{
    if (head == null && tail== null)
        head = tail = data;

    else {
        data.prev=tail;
        tail.next=data;
        tail=data;}
        size++;
    public void addIndex(Node data,int pos)
    {
        if (size>pos && pos>0) {
            Node current = head;
            for(int i=1; i < size-1  ; i++)
            {current=current.next;}

            Node newNext = current.next;
            current.next=data;
            data.prev=current;
            data.next= newNext;
            newNext.prev=data;
            size++;


            }
        else System.out.println("error");
    }

    public void removeFirst()
    {
        if (head != null && tail != null)
        {
            head=head.next;
            size--;
        }
    }

    public void removeLast() {
        if (head != null && tail != null)
        {
            Node temp= tail.prev;
            tail=tail.prev;
            tail.prev=temp;
            tail.next=null;
            size--;
        }
    }
public void removeIndex(int pos)
{
    if (size>pos && pos>0) {
        Node current = head;
        for(int i=1; i < pos  ; i++)
            current=current.next;

        Node delete = current;
        delete.prev.next=delete.next;
        delete.next.prev=delete.prev;
        size--;

    }}

public void view() {

    Node current = head;
    while(current !=null) {
        current.displayNode();
        current = current.next;}
    System.out.println();
    ;}
```

Node

```
public class Node
{
    int data;
    Node next;
    Node prev;

    public Node(int data)
    {
        this.data=data;
        next = null;
        prev = null;
    }

    public void displayNode() {
        System.out.print(data+"    ");
    }}
```

הפיכת רשימה מקושרת (במקום, ללא מבנה נתונים נוסף):

הרעיון הוא לבצע הפיכה של המצביעים מהסוף להתחלה עם מצביעי עזר:

```
Link current = head;  //next,prev = null
while (current != null) {
  next = current.next;
  current.next = prev;
  prev = current;
  current = next;}
```

בסוף התהליך (אחרי סיום הלולאה) נוודא שhead מצביע על prev (שהוא עכשיו האיבר הראשון): head=prev;

בידקה אם קיים איבר בlinklist

```
boolean member( int k, Node p ) {
if (p == null) return false;
else if( k == p.item ) return true;
else return member( k, p.next );    }
```

מימש תור ברשימה מקושרת

```
class LinkedQueue <Item> implements
Queue <Item>{
  private Node front, rear;
  private int size;
  private class Node{
    Item item;
    Node next;}
  public LinkedQueue() {
    front = null;
  rear = null;
    size = 0;}
  public boolean isEmpty() {
    return (size == 0);}
  public Item delete() {
    Item item = front.item;
    front = front.next;
    if (isEmpty())
      rear = null;
    size--;
    return item;}
  public void insert(Item item){
    Node oldRear = rear;
    rear = new Node();
    rear.item = item;
    rear.next = null;
    if (isEmpty())
      front = rear;
    else
      oldRear.next = rear;
    size++;}
  public int size() {
    return size; }}
```

רשימה מקושרתמימוש מחסנית

```
public class StackUsingLinkedlist {
private class Node {
int data;
Node link;  }
Node top;
StackUsingLinkedlist()
{ this.top = null; }
public void push(int x) {
Node temp = new Node();
if (temp == null) {
System.out.print("null");
return; }
temp.data = x;
temp.link = top;
top = temp; }
public boolean isEmpty() {
return top == null; }
public int peek()  {
if (!isEmpty()) {
return top.data; }
else {
System.out.println("Stack is empty");
return -1;}}
public void pop(){
if (top == null)
return;
```

שאלת הפילנדרום

```
void reverse()  {
    Node prev = null;
    Node current = second_half;
    Node next;
    while (current != null)  {
        next = current.next;
        current.next = prev;
        prev = current;
        current = next; }
    second_half = prev; }
```

מימוש של רשימה מקושרת חד כיוונית

```
public class LinkedList
{
    Node head;
    Node tail;
    int size;

    public LinkedList() {
        this.head = null;
        this.tail = null;
        this.size = 0;
    }

    public void addFirst(Node data)
    {
        if (size==0)
        {
            head=data;
            tail=data;
            this.size++;
        }
        else
        {
            Node current = head;
            head=data;
            head.next=current;
            this.size++;
        }   }

    public void addLast(Node data)
    {
        if (head == null)
        {
            head=data;
            tail=data;
        }
        else
        {
            tail.next=data;
            tail=data;
        }
        this.size++;}


    public void addAfterIndex(Node data,int index)
    {
        if(index > size)
            System.out.println("error");
        else
        {
            Node current = head;

            for (int k = 1; k < index; k++)
            {
                current = current.next;
            }
            Node temp= current.next;
            current.next = data;
            data.next=temp;
            size++;
        }
    }
    public void removeFirst()
    {
        if(tail == head)
            System.out.println("error");
        else
        {
            head=head.next;
            size--;
        }
    }

    public void removeLast()
    {
        if(tail == head)
            System.out.println("error");
        else
        {
            Node current = head;
            for (int i = 1; i < size-1; i++)
            {
                current=current.next;
            }
            current.next=null;
            tail=current;
        }
    }
    public void removeIndex(int index) {
        if(tail == head)
            System.out.println("error");
        else
        {
            Node current = head;
            for (int i = 1; i < index-1; i++)
                current=current.next;

            current.next=current.next.next;
            size--;
        }
    }
```