L2: Verkettete Liste (dynamisch allokiert)

Bekommen in L3

Abgabe in L4

Implementiere in C++ den gegebenen **Container (ADT)** mithilfe der gegebenen **Repräsentierung** und mit einer **verketteten Liste dynamisch allokiert (**SLL – Simple Linked List, DLL – Double Linked List**)** als Datenstruktur. Die verkettete Liste muss man selber implementieren. Für die Implementierung dürft ihr keine Containers oder Datenstrukturen aus STL (oder aus anderen Bibliotheken) benutzen.

Bemerkung. Für Repräsentierung, die nicht „standard“ sind, bitte siehe Vorlesung für Beispiele!

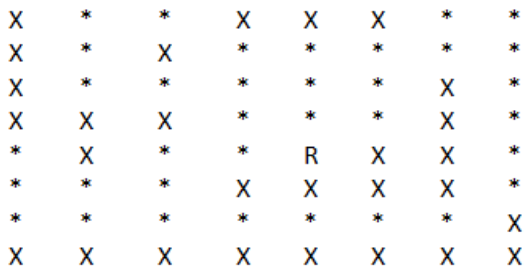
1. **ADT Matrix** – repräsentiert als schwachbesetzte Matrix (sparse), indem man ein SLL von Tupeln der Form (Zeile, Spalte, Wert) (Wert $\neq 0$) benutzt, wobei die Tupel in lexikographischen Reihenfolge nach (*Zeile*, *Spalte*) gespeichert werden.
2. **ADT Matrix** – repräsentiert als schwachbesetzte Matrix (sparse), indem man ein DLL von Tupeln der Form (Zeile, Spalte, Wert) (Wert $\neq 0$) benutzt, wobei die Tupel in lexikographischen Reihenfolge nach (*Zeile*, *Spalte*) gespeichert werden.
3. **ADT Bag** – repräsentiert mithilfe einer SLL von Paaren der Form (*Element*, *Frequenz*).
4. **ADT Bag** – repräsentiert mithilfe einer DLL von Paaren der Form (*Element*, *Frequenz*).
5. **ADT SortedBag** – mit Elementen vom Typ **TComp** repräsentiert mithilfe einer SLL von Paaren der Form *(Element*, *Frequenz)*, sortiert mithilfe einer Relation auf den Elementen
6. **ADT SortedBag** – mit Elementen vom Typ **TComp** repräsentiert mithilfe einer DLL von Paaren der Form *(Element*, *Frequenz)*, sortiert mithilfe einer Relation auf den Elementen
7. **ADT SortedSet** – mit Elementen vom Typ **TComp** repräsentiert mithilfe einer SLL, sortiert mithilfe einer Relation auf den Elementen
8. **ADT SortedSet** – mit Elementen vom Typ **TComp** repräsentiert mithilfe einer DLL, sortiert mithilfe einer Relation auf den Elementen
9. **ADT Set** – repräsentiert mithilfe einer SLL von Elementen
10. **ADT Set** – repräsentiert mithilfe einer DLL von Elementen
11. **ADT IndexedList** – repräsentiert mithilfe einer SLL
12. **ADT IndexedList** – repräsentiert mithilfe einer DLL
13. **ADT IteratedList** – repräsentiert mithilfe einer SLL
14. **ADT IteratedList** – repräsentiert mithilfe einer DLL
15. **ADT Map** – repräsentiert mithilfe einer SLL von Paaren der Form (*key*, *value*)
16. **ADT Map** – repräsentiert mithilfe einer DLL von Paaren der Form (*key*, *value*)
17. **ADT MultiMap** – repräsentiert mithilfe einer SLL von Paaren der Form (*key*, *value*). Ein Schlüssel kann mehrmals vorkommen.
18. **ADT MultiMap** – repräsentiert mithilfe einer DLL von Paaren der Form (*key*, *value*). Ein Schlüssel kann mehrmals vorkommen.

19. **ADT MultiMap** – repräsentiert mithilfe einer SLL mit eindeutigen Schlüsseln (key). Für jeden Schlüssel speichert man eine SLL von Werten (value).
20. **ADT MultiMap** – repräsentiert mithilfe einer DLL mit eindeutigen Schlüsseln (key). Für jeden Schlüssel speichert man eine DLL von Werten (value).
21. **ADT SortedIndexedList** – repräsentiert mithilfe einer SLL sortiert mithilfe einer Relation auf den Elementen
22. **ADT SortedIteratedList** – repräsentiert mithilfe einer SLL sortiert mithilfe einer Relation auf den Elementen
23. **ADT SortedIndexedList** – repräsentiert mithilfe einer DLL sortiert mithilfe einer Relation auf den Elementen
24. **ADT SortedIteratedList** – repräsentiert mithilfe einer DLL sortiert mithilfe einer Relation auf den Elementen
25. **ADT PriorityQueue** – repräsentiert mithilfe einer SLL von Paaren (Element, Priorität) sortiert nach den Prioritäten mithilfe einer Relation
26. **ADT PriorityQueue** – repräsentiert mithilfe einer DLL von Paaren (Element, Priorität) sortiert nach den Prioritäten mithilfe einer Relation
27. **ADT SortedMap** – repräsentiert mithilfe einer SLL von Paaren der Form (*key*, *value*) und sortiert mithilfe einer Relation auf den Schlüsseln (key)
28. **ADT SortedMap** – repräsentiert mithilfe einer DLL von Paaren der Form (*key*, *value*) und sortiert mithilfe einer Relation auf den Schlüsseln (key)
29. **ADT SortedMultiMap** – repräsentiert mithilfe einer SLL mit eindeutigen Schlüsseln (key) und sortiert mithilfe einer Relation auf den Schlüsseln (key). Für jeden Schlüssel speichert man eine SLL von Werten (value).
30. **ADT SortedMultiMap** – repräsentiert mithilfe einer DLL mit eindeutigen Schlüsseln (key) und sortiert mithilfe einer Relation auf den Schlüsseln (key). Für jeden Schlüssel speichert man eine DLL von Werten (value).
31. **ADT SortedMultiMap** – repräsentiert mithilfe einer SLL von Paaren der Form (*key*, *value*) und sortiert mithilfe einer Relation auf den Schlüsseln (key). Ein Schlüssel kann mehrmals vorkommen.
32. **ADT SortedMultiMap** – repräsentiert mithilfe einer DLL von Paaren der Form (*key*, *value*) und sortiert mithilfe einer Relation auf den Schlüsseln (key). Ein Schlüssel kann mehrmals vorkommen.

---

33. **ADT Matrix** – repräsentiert als schwachbesetzte Matrix (sparse) mit zirkulären Listen verbunden miteinander.

34. **Roboter in dem Labyrinth:**
    In einem rechteckigen Labyrinth gibt es besetzte (X) und leere Zellen (*). Der Roboter kann sich in vier Richtungen bewegen: Nord, Süd, Ost, West.

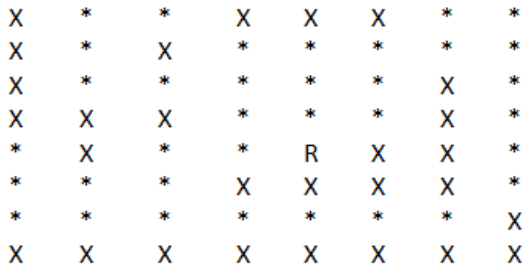


a) Bestimme ob es einen Pfad gibt, sodass der Roboter aus dem Labyrinth rauskommt (d.h. zu der ersten oder letzten Spalte oder zu der ersten oder letzten Zeile gelangen).
b) Bestimme einen Pfad, falls es einen gibt.
c) Bestimme einen Pfad mit minimaler Länge, falls es einen gibt.

Ihr müsst in der Implementierung **ADT Queue** repräsentiert als **SLL** benutzen.

35. Roboter in dem Labyrinth:

In einem rechteckigen Labyrinth gibt es besetzte (X) und leere Zellen (*). Der Roboter kann sich in vier Richtungen bewegen: Nord, Süd, Ost, West.



d) Bestimme ob es einen Pfad gibt, sodass der Roboter aus dem Labyrinth rauskommt (d.h. zu der ersten oder letzten Spalte oder zu der ersten oder letzten Zeile gelangen).
e) Bestimme einen Pfad, falls es einen gibt.
f) Bestimme einen Pfad mit minimaler Länge, falls es einen gibt.

Ihr müsst in der Implementierung **ADT Queue** repräsentiert als **DLL** benutzen.

36. Rot-Schwarz Kartenspiel:

Zwei Spieler bekommen je n/2 Karten (rot und schwarz). Die zwei Spieler legen der Reihe nach je eine Karte (die oberste Karte aus ihrem Stapel) auf den Ablagestapel. Falls ein Spieler eine rote Karte gelegt hat, dann nimmt der andere Spieler alle Karten von dem Ablagestapel und legt diese am Ende seines Stapels. Das Spiel ist beendet, wenn einer der Spieler alle n Karten in der Hand hat.

Simuliere das Spiel für n Karten.
Ihr müsst in der Implementierung **ADT Stack** repräsentiert mithilfe einer **SLL** und **ADT Queue** repräsentiert mithilfe einer **DLL** benutzen.

37. **Rot-Schwarz Kartenspiel:**
    Zwei Spieler bekommen je n/2 Karten (rot und schwarz). Die zwei Spieler legen der Reihe nach je eine Karte (die oberste Karte aus ihrem Stapel) auf den Ablagestapel. Falls ein Spieler eine rote Karte gelegt hat, dann nimmt der andere Spieler alle Karten von dem Ablagestapel und legt diese am Ende seines Stapels. Das Spiel ist beendet, wenn einer der Spieler alle n Karten in der Hand hat.
    Simuliere das Spiel für n Karten.
    Ihr müsst in der Implementierung **ADT Stack** repräsentiert mithilfe einer **DLL** und **ADT Queue** repräsentiert mithilfe einer **SLL** benutzen.

38. **Auswertung eines arithmetischen Ausdruckes in Infixnotation** (der Ausdruck enthält Klammern). Man muss die Infixnotation **in Postfixnotation umwandeln** und dann die Postfixnotation auswerten.
    Ihr müsst in der Implementierung **ADT Queue** repräsentiert mithilfe einer **SLL** und **ADT Stack** repräsentiert mithilfe einer **DLL** benutzen.

39. **Auswertung eines arithmetischen Ausdruckes in Infixnotation** (der Ausdruck enthält Klammern). Man muss die Infixnotation **in Postfixnotation umwandeln** und dann die Postfixnotation auswerten.
    Ihr müsst in der Implementierung **ADT Queue** repräsentiert mithilfe einer **DLL** und **ADT Stack** repräsentiert mithilfe einer **SLL** benutzen.