Luleå tekniska universitet

Skellefteå

**Laborationsrapport**

D0037D Objektorienterad Programmering

Laboration 1

**Pontus Stenlund**

**ponste-5@student.ltu.se**

Luleå Tekniska Universitet

Skellefteå

2016-02-05

Innehållsförteckning

**Problemspecifikation3**

**Användarhandledning4**

**Algoritmbeskrivning5**

**Systembeskrivning6**

LinkedList6

LinkedList\_Arr7

Stack8

Queue9

**Lösningens begränsningar10**

**Diskussioner11**

**Testkörningar12**

**Problemspecifikation**

I del 1 av uppgiften så skulle vi implementera en vanlig enkellänkad lista som skulle följa ADT som beskrev vad varje funktion skulle ha för körningsresultat.

I del 2 så skulle vi implementera vår enkellänkade lista som en template och i en array. Vi skulle även implementera en Stack och Queue.

I del 3 så var vår uppgift att skriva ett program som testkörde koden i del 1 och del 2. Vi skulle först använda programmet på vår egen kod och sedan på en av våra klasskamraters kod.

**Användarhandledning**

För att skapa en objekt av LinkedList från del 1 så är det bara att skriva LinkedList "variabelnamn" och anropa de funktioner som finns. Programmet lägger automatiskt till noder och länkar ihop dom vid anrop av insertBefore(). Och programmet länkar automatiskt om noderna vid anrop av removeAt(). Datan som kan sparas här är enbart integers.

LinkedList\_Arr, Stack och Queue är alla template klasser så det är viktigt att ange vilken typ man vill ska sparas.

LinkedList\_Arr har samma funktioner som i LinkedList från del 1.

Stacken har funktioner som Push(T in\_data) som lägger till en nod med data. Pop() tar bort noden. Empty\_Check() kollar ifall det finns några noder i Stacken och returnerar true/false. Get\_Top() returnerar värdet längst upp i stacken (dvs det som senast lades dit). Get\_Top() kastar även ett range\_error ifall man försöker hämta data från Stacken men den är tom.

Queue har funktioner som Enqueue(in\_data) som lägger till en nod längst bak i kön. Dequeue tar bort det värde som ligger längst fram i kön. Get\_Front() returnerar värdet som ligger längst fram.

För att använda min unit tester så skapar man först ett objekt av LLtester genom att använda LLtester<"typ"> "variabelnamn". Sedan så skickar man in LinkedList, LinkedList\_Arr, Stack eller Queue som argument till de funktionerna. Den funktion som testar det mesta är Start\_Test("klass"). En sak som är viktig att veta är att typen på LLtester och de andra objekten måste vara samma annars kommer inte programmet att kompilera.

**Algoritmbeskrivning**

Den algoritm som jag använder mig av är linjär sökning.

Detta är att eftersom noderna i LinkedList vet bara vad om är framför och inte bakom så därför går det inte att göra på något annat sätt.

Stack och Queue's noder håller reda på det som är bakom men inte framför så även där så måste vi använda oss av linjär sökning.

**Systembeskrivning**

**LinkedList**

Enkellänkad lista fungerar som sådan att man har en klass som har en struct eller klasser som Noder som sparar data.

Node klassen har bara har en variabel som sparar data och en pekarvariabel, t.ex. next, som ska peka till nästa nod.

Länkade listan måste ha åtminstone tre nodpekare, t.ex. iteratorpos, iterator\_pre, och head.

Det är lättast att i konstruktorn skapa en ny nod (dummy node) och låta head peka på den. Och sedan sätta iteratorpos = head->next så att den inte pekar på samma nod som head. Och sedan sätta iterator\_pre till head.

Sen har man ett antal metoder som isOff, item, start, forth, insertBefore och removeAt .

isOff kollar ifall iteratorpos == nullptr och returnerar true eller false. Den här metoden används i item, forth och removeAt.

Item anropar isOff och om isOff returnerar false så kör vi return på nodens data.

Start pekar om iteratorpos till head->next och iterator\_pre till head. Dvs vi går tillbaka till början av listan.

Forth pekar om iterator\_pre till samma nod som iteratorpos pekar på och sedan pekar man om iteratorpos till iteratorpos->next.

InsertBefore skapar en nod sätter en nod mellan iteratorpos och interator\_pre, lägger till värdet som man skickar med som argument och länkar om iterator\_pre till den nya noden. IteratorPos pekas aldrig om här.

removeAt skapar en ny nodpekare (temp) som pekar på samma nod som iteratorpos. Sedan pekar vi om iteratorpos till iteratorpos-next och iteratorpre-next till current. Sedan kör vi delete på den nod som den temp pekar på.

**LinkedList\_Arr**

För del 2 i labben så använder vi oss en array som vi allokerar noder till. I konstruktorn så sätter vi storleken på arrayen. Sedan har vi samma funktioner som i den enkellänkade listan.

Men nu har vi inte pekare som som pekar på noder. Nu har vi bara en arraypekare som pekar på första elementet i listan och ett par int variabler som får värdet på det index som t.ex iteratorpos är.

Det blir lite problematiskt att hålla reda på vilka noder som är lediga att tilldela data till så min lösning är att ha en stack som håller reda på de index som är lediga att användas.

Så om vi anropar insertBefore så kollar vi först i stacken om det finns ett index att hämta, hämtar det, tilldelar data till det indexet, sätter nodens next till iteratorpos index. Sedan sätter vi iterator\_pre till index. Sedan så tar vi bort datan vi nyss hämtat från stacken. Men här får jag ett par specialfall på grund av att jag inte använder mig av en dummy node. Om våra ”pekare” har värder -1 så är dom utanför listan. Så om t.ex iterator\_pre == -1 så antyder det att vi vill sätta om head till det nya indexet. Annars sätter vi bara iterator\_pre.next till indexet vi får från stacken.

Men eftersom jag använder mig av en stack för att hålla reda på vilka index som är lediga så har stacken en funktion som kollar ifall den är tom. Så om stacken är tom och vi försöker att hämta ett nytt index så kastar den ett range\_error. Så eftersom jag inte vill att programmet ska krascha så har jag en try, catch där som gör att programmet inte kraschar när vi försöker lägga till mer data än vad arrayen är satt till.

**Stacken**

För stacken så har jag en klass med noder. Noden ser ut exakt som i LinkedList förutom att jag har gjort den som en template vilket gör att vi kan sätta in vilket typ som helst.

Sedan har jag bara en nodpekare som alltid ska peka längst upp på stacken.

Sedan så har jag metoder som Push, Pop, Empty\_Check och Get\_Top.

Push skapar en ny nod, tilldelar datan, och pekar om new\_top->next till top. Sedan pekar jag om top till new\_top.

Pop skapar en temp pekare som vi pekar till top. Sedan pekar vi om top till den nod som ligger under den. Sedan kör vi delete på temp.

Empty\_Check kollar ifall top == nullptr och returnerar true/false.

Get\_Top returnerar det värde på noden som top pekar på så länge som top != nullptr. Isåfall så kastar vi ett range\_error som säger att stacken är tom.

**Queue**

I queue så använder jag mig ännu en gång av noder. Noden ser exakt likadan ut som i stacken. Men nu har jag två nodpekare i klassen Queue.

Detta är för att hålla reda på det som är längst fram i kön och det som är längst bak.

Metoderna som finns i denna klass är Enqueue, Dequeue och Get\_Front.

Enqueue kollar först ifall nodpekaren front är == nullptr för då allokerar vi en ny nod som front får peka på, tilldelar data till den noden och sätter nodpekaren back = front.

Men om front redan pekar på en nod så allokerar vi en ny nod som vi döper till temp. Tilldelar värdet till noden och sedan så sätter vi back->next till temp. Och sedan pekar vi om back till samma objekt som temp.

Dequeue kollar först ifall front är == nullptr. Om den är det så händer ingenting. Men om den inte är == nullptr så skapar vi ännu en gång en temp nodpekare som vi låter pekar på samma nod som front. Sedan sätter vi om front till front->next. Sedan måste vi kolla ifall back == temp. Då pekar vi om back till nullptr. Sedan när allt är pekat om så kör vi delete på temp.

Get\_Front kollar ifall front == nullptr. Om den inte är det så returnerar vi det värde på noden som front pekar på. Annars kastar vi ett range\_error som säger att queuen är tom.

**Lösningens begränsningar**

De begränsningar som jag vet är att i LLtester bara tar emot klasser med samma typ som sig själv. T.ex om jag skapar ett objekt av LLtester med typen <int> så kan jag inte skicka in en Stack med typen <float>.

Sedan så i LinkedList\_Arr så använder jag mig av en Stack för att hålla reda på de index på arrayen som är lediga. Och min Stack har en funktion Get\_Top() som anropar Empty\_Check() som kollar ifall stacken är tom. Är stacken tom när Get\_Top() anropas så kastas std::range\_error("The stack is empty"). Detta blir då ett specialfall för mig så i LinkedList\_Arr.insertBefore(data) så fångar jag det som blir kastat från Stack. Detta kan innebära att man får ett felmeddelande fast man inte förväntar sig det.

Sedan så när man kör LLtester.Start\_Test("klass") så borde inte det returneras något värde efter ett tag eftersom jag har en funktion som anropas som tar bort alla noder + 1 eftersom jag vill testa att ta bort fler noder än det som finns. Efter det så testar jag att hämta ett värde från en tom lista. Då skrivs antingen 0 eller ett slumpmässigt tal vilket jag antar är en pekaradress.

Det är de begränsningar som jag kan komma på i skrivandets stund.

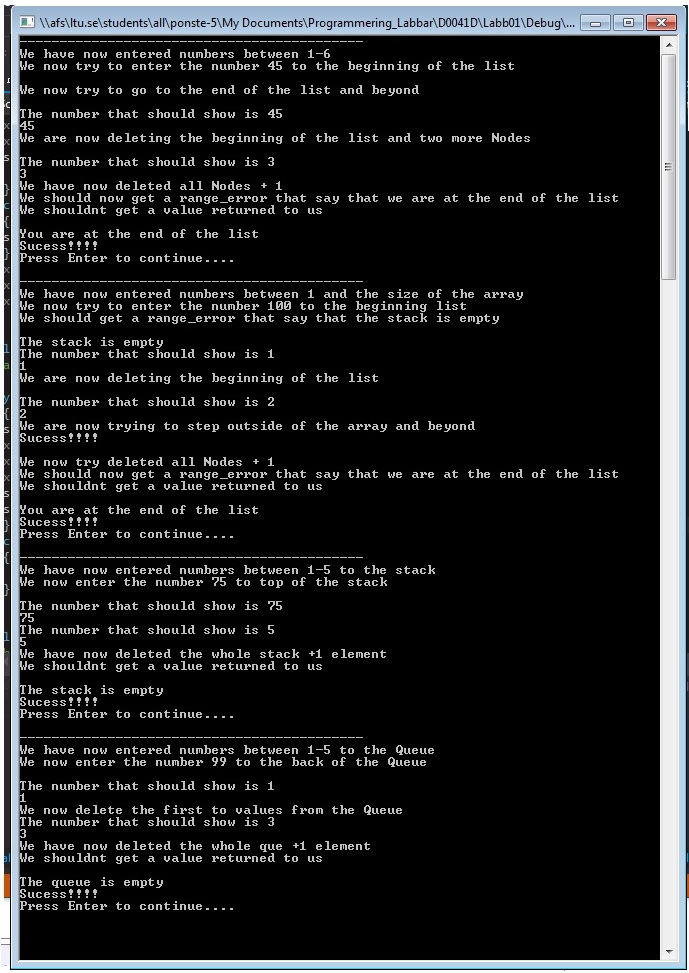
**Diskussion**

I den här labben så känner jag att jag har blivit mer bekväm med att använda pekare och att hålla koll på vart i koden jag kan få minnesläckor.

Och i denna laboration så har jag insett att det är riktigt användbart att rita ner hur programmet ska fungera.

Tycker att det var rätt roligt att det var en lite mer abstrakt labb. Det gjorde så att man blev tvungen att tänka efter lite extra på vad som skulle hända och hur man skulle kunna implementera det.

**Testkörningar**

Mitt test på min kod

Testkörning på Hans kod

