Linköpings Universitet
Institutionen för datavetenskap (IDA)
UPP-gruppen 2017-01-16

Pekare och dubbellänkad lista

Mål

I den här laborationen skall du lära dig hantera pekare och dynamiskt minne. Speciellt ska du lära dig hur konstruktorer, destruktorer och operatorer i en klass kan hjälpa till med ansvaret att hålla reda på pekare och allokerat minne i en klass.

Läsanvisningar

- Pekare
  - Adressoperatorn &
  - Innehållsoperatorn, avreferering med *
  - Piloperatorn, medlemsåtkomst med ->
- Inre klasser
- Speciella medlemsfunktioner
  - Konstruktorer (defaultinitiering eller från andra värden/typer)
  - Kopieringskonstruktor (djup kopiering till nytt objekt)
  - Kopieringstilldelning (djup kopiering till befintligt objekt)
  - Flyttkonstruktor (snabb flytt från döende till nytt objekt)
  - Flytttilldelning (snabb flytt från döende till befintligt objekt)
  - Destruktor (djup destruktion)

Uppgift: Dubbellänkad sorterad lista

Du skall skapa en rak dubbellänkad sorterad lista innehållandes heltal. Det skall finnas en klass som representerar hela listan, komplett med funktioner för att hantera kopiering av hela listan korrekt i alla lägen. Klassen ska garantera korrekt minnesanvändning och förhindra minnesläckage.

Internt, som en inre klass, skall varje länk i listan byggas upp av ett eget klassobjekt som håller reda på det lagrade värdet och en pekare till nästa länk och föregående länk. Det är dessa länkobjekt som tillsammans bygger upp en kedja som formar listan som helhet.
Själva listklassen behöver endast innehålla en pekare till den första länken, men problemlösningen kan göras bättre om den även innehåller en pekare till sista länken. Dessutom behöver klassen *som minst* funktioner för sorterad insättning av nya värden (dubbletter tillåts), borttagning och åtkomst av ett specifikt index samt utskrift av hela listans innehåll på en given utskriftsström. Om ett index inte finns i listan ska ett lämpligt undantag genereras.

För att kunna använda (testa) listan på ett vettigt sätt kommer du behöver lite fler funktioner än minimikraven ovan. Tänk på om alla dessa ska vara synliga eller om några bara är relevanta för att säkerställa att klasserna fungerar korrekt.

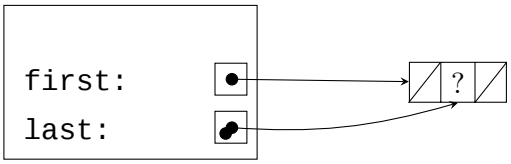
Listan ska kunna skapas antingen tom eller med givna startvärden enligt följande exempel:

```
Sorted_List empty_list;                 // tom lista skapas
Sorted_List initialized_list{2,7,5};    // lista med värdena 2,5,7
```

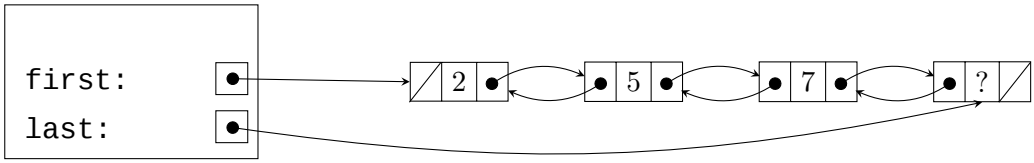
Programmeraren som använder listklassen skall inte någonsin behöva veta (eller bry sig om) hur listan är uppbyggd internt. Länkklassen bör alltså hållas privat och oåtkomlig, alternativt oanvändbar, för allt utom listklassen.

Alla funktioner du skriver ska testas med testfall som täcker in alla undantagsfall som kan inträffa, exempelvis kan insättning först i en lista behöva uppdatera andra pekare än insättning mellan två befintliga länkar. Fler sådana fall kan finnas beroende på lösnigsmetod och använda algoritmer. Det krävs att du utöver din klass även redovisar ditt testprogram.
Under arbetet rekommenderas att du använder dig av testdriven utveckling (TDD). Även om TDD inte används ska du skriva testfall för att se till att dina lösningar faktiskt fungerar.

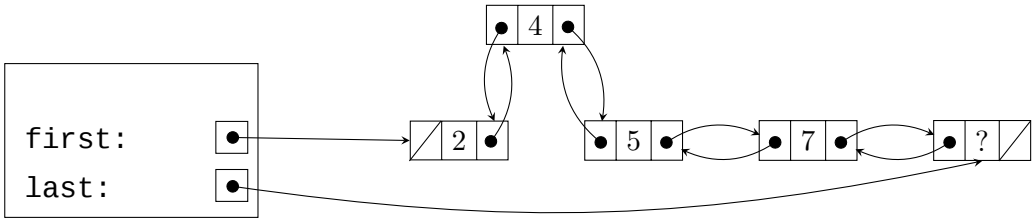
För att undvika flera specialfall bör listan alltid ha minst en länk, en så kallad sentinel. Värdet i denna nod är ej av betydelse och bör aldrig läsas. Vi sätter därför endast **?** som värde i de följande figurerna för att markera sentinelnoden. En tom lista innebär då att listans båda pekare pekar på samma länk (se figur 1).
Du bör börja med att tänka igenom hur insättning och borttagning ska fungera och rita operationerna på papper. Det är lämpligt att börja med en tom lista, stegvis sätta in $5, 3, 9, 7$, ta bort dem i samma ordning och för varje steg fundera på vilka pekarvariabler som måste uppdateras och i vilken ordning. Figur 2 ritar ut en lista som den ser ut efter insättning av 5, 2 och 7. Figur 3 visar hur listan ser ut efter insättning av även 4.



Figur 1: Tom lista



Figur 2: Lista innehållandes värdena 2,5 och 7



Figur 3: Lista i figur 2 efter insättning av 4

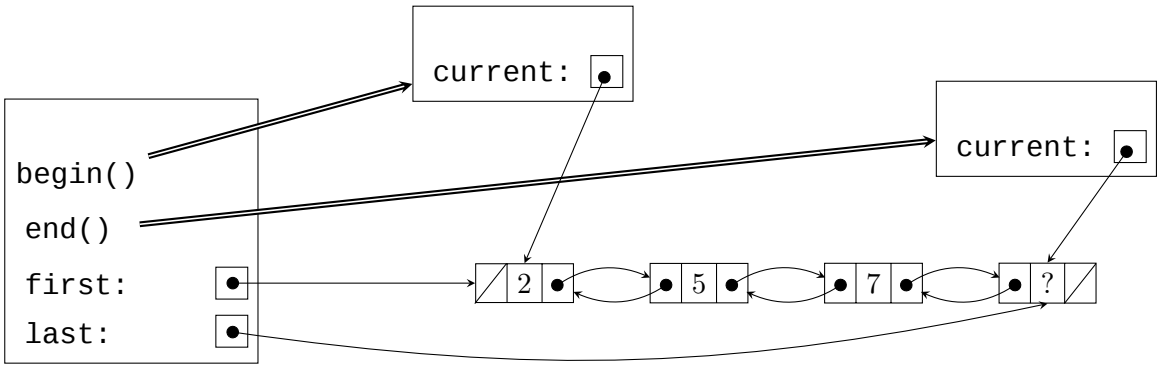
Givetvis skall alla fyra operationer (insättning, borttagning, åtkomst och utskrift) fungera korrekt på alla listor, inklusive tomma. Likaså skall kopiering och tilldelning av listor fungera och skapa djupa kopior (ändringar i kopian påverkar inte originalet). Även flyttsemantik ska stödjas av din klass. Ditt testprogram skall kontrollera dessa operationer.

Bonusuppgift: Smart stegning genom listan

När din funktion för att hämta ut ett specifikt element anropas så kommer du gå igenom hela listan fram till det elementet. Om du då tänker dig att din funktion används för att stega igenom listan från start till slut kommer man ju börja om från början för varje element. Det är inte så effektivt. Det vore smart om man kunde spara undan var i listan man är och i nästa iteration bara gå ett steg framåt (till nästa länk).
Du ska implementera en klass som håller reda på var man är (vilken länk man är på) och tillåter att man går ett steg framåt i listan. För att få ut ett objekt av din nya klasstyp ska man kunna fråga listan om ett sådant genom att anropa medlemsfunktionen `begin`. Begin kommer då att ge oss ett objekt som pekar ut början på listan. För att kunna avgöra när man kommit till slutet ska man också kunna be listan om ett objekt av denna typ som anger att det inte finns fler länkar. Denna medlemsfunktion ska heta `end`. När du är klar ska kodexempel 1 skriva ut samtliga värden i listan. Tänk i förväg igenom vilka operatorer din klass kommer behöva för att kodexemplet ska fungera. Observera att `auto` här kan ersättas med namnet på din nya klass.

```
Sorted_List lista{2, 5, 7};
for ( auto it = lista.begin(); it != lista.end(); ++it)
{
  cout << *it << endl;
}
```

Kodexempel 1: Kodexempel för extrauppgiften



Figur 4: Extra objekt för smart stegning, dubbelpilen ska tolkas som att funktionen returnerar objektet den pekar på (det är inte en pekare)