Mittuniversitetet
MID SWEDEN UNIVERSITY

Laboration 2 - Länkade listor
Objektbaserad programmering i C++

Jimmy Åhlander*

20 mars 2025

Innehåll

1 Introduktion

Det finns många olika datastrukturer att välja mellan när vi programmerar. En sådan som vi tidigare arbetat med är `vector`. En vector är en behållare (*container*) med en array som underliggande datastruktur. En `string` är också en mer specialiserad behållare som håller en array av `char`.

*jimmy.ahlander@miun.se. Institutionen för data- och elektroteknik (DET).
Baserad på laborationen *Länkade listor* av Martin Kjellqvist.

Precis som alla andra datastrukturer medför även arrayer olika för– och nackdelar. Åtkomsten till element blir omedelbar i en vector men det är istället kostsamt att förändra storleken på den. När en vector växer eller krymper krävs en omallokering och kopiering av samtliga element till en ny vector av den önskade storleken. Lite som när en eremitkräfta växt ur sitt skal; skalet kan inte växa så kräftan måste hitta ett nytt, större skal att flytta in i. Precis som eremitkräftan väljer sitt nya skal omsorgsfullt för att det ska finnas lite extra utrymme att växa i så tillämpar även `vector` och `string` olika strategier för att minska tillväxtskostnaden. Tumregeln är att öka kapaciteten med 50 %. Det vill säga, har vi en kapacitet på 10 element och är påväg att lägga till ett elfte så ökar vi kapaciteten till 15. Detta är ett klassiskt exempel på **space-time tradeoff** som återkommer ofta i datateknik, där vi potentiellt slösar minne för den kapacitet som inte nyttjas avvägt mot den tid vi tjänar på att inte behöva omallokera alla element varje gång vectorn växer. Eremitkräftan tänker också att det är bättre att bära runt ett lite för stort skal än att behöva byta varje dag.

Många gånger önskar vi oss istället datastrukturer som kan förändra sin storlek utan denna extra kostnad. De kallas för **dynamiska datastrukturer**. En vector är förvisso dynamisk från programmerarens perspektiv, men den arbetar med en statisk underliggande datastruktur. Dynamiska datastrukturer nyttjar dynamiskt allokerat minne på heapen och pekare för att organisera sin data. Exempel på sådana strukturer är stack och kö som i sin tur oftast har en länkad lista som underliggande datastruktur. I den här laborationen kommer du att implementera en länkad lista.

2 Mål

Efter genomförd laboration ska du kunna skapa och använda klasser som nyttjar pekare och dynamiskt allokerar minne, exempelvis för dynamiska datastrukturer.

Följande lärandemål är kopplade till laborationen. Du ska kunna:

- använda pekare i praktisk programmering och i samband med dynamisk minnesallokering.
- utnyttja iteratorer i standardbiblioteket.
- redogöra för skillnaden mellan djup och ytlig kopiering och kunna överlagra nödvändiga operatorer i samband med detta.
- specificera och implementera någon klassisk abstrakt datastruktur som en klass samt använda denna.
- utnyttja konstanta typer för medlemmar, objektreferenser och pekare.

I modern C++ är behållarna i STL så pass optimerade att vi knappast kan hoppas på att göra ett bättre jobb själva. I STL finns bland annat `std::list`, en dubbellänkad lista som löser vår uppgift utmärkt. Syftet med laborationen

är därför att implementera en länkad listad för att undersöka datastrukturer, inte för att vi behöver en ny variant.

3 Teori

En övergripande beskrivning av de olika behållarna i C++ hittar du i [1, s. 326]. De vanligaste operationerna på dessa behållare beskrivs vidare i [1, s. 330]. Inför den här laborationen bör du vara införstådd i hur klasser fungerar, hur de konstrueras och destrueras. Det är centralt att du förstår de olika typerna av konstruktorer som finns, bland annat den parameterlösa konstruktorn och kopieringskonstruktorn. Du kommer även behöva överlagra tilldelningsoperatorn och en operator för konkatenering. Du bör ha läst om dynamisk minnesallokering, pekare och länkade listor då listans noder kommer att skapas dynamiskt och lagras på heapen.

Följande begrepp används i laborationen:

länkad lista En lista bestående av noder som sammanlänkas med varandra.

dubbellänkad lista En länkad lista där noderna innehåller två länkar vilket möjliggör traversering både fram- och baklänges i listan.

node En `struct` som innehåller data för ett element i listan och länkar till andra noder.

länk En pekare till en annan nod. Två länkar finns i varje nod: `next` och `prev`.

next En pekare som leder till nästa nod i listan närmare svansen.

prev En pekare som leder till föregående nod i listan närmare huvudet.

head En pekare till den första noden i listan.

tail En pekare till den sista noden i listan.

4 Uppgift

Din uppgift är att implementera en dubbellänkad lista enligt följande API.

```
class linked_list {
  public:
  linked_list();
  linked_list(const linked_list &src);

  ~linked_list();

  linked_list& operator=(const linked_list &rhs);

  // appends elements from rhs
  linked_list& operator+=(const linked_list &rhs);

  // inserting elements
  void insert(double value, size_t pos);
  void push_front(double value);
  void push_back(double value);

  // accessing elements
  double front() const;
  double back() const;
  double at(size_t pos) const;

  // removing elements
  void remove(size_t pos);
  double pop_front();
  double pop_back();

  // status
  size_t size() const;
  bool is_empty() const;

  // output
  void print() const;
  void print_reverse() const;

  private:
  struct node {
    node(double value); // konstruktor för noden
    double value;
    node* next;
    node* prev;
  };
  node* head;
  node* tail;
};
```

Din lista ska alltså bestå av noder som håller flyttal. Observera att structen `node` är inkapslad och deklarared inom scopet för `linked_list`. Det innebär att noder inte kan konstrueras ute i det vilda utan endast inom den länkade listans metoder. För att implementera konstruktorn till noden måste du därför använda scopingoperatorn två gånger på följande sätt:

```
linked_list::node::node(double value) { /* implementation */ }
```

När din lista är implementerad ska du testa funktionaliteten i main enligt följande:

1. `push_back`

   Fyll två länkade listor med 10 stigande slumpade heltal vardera med start vid 0. Varje efterföljande tal ska vara 0–10 större än föregående tal.

2. `at` och `remove`

   Kontrollera vilken lista vars 5:e element är störst och ta bort det. Är de lika stora tar du bort från valfri lista.

3. `operator=`

   Deklarera en tredje lista och tilldela därefter den minsta listan av de två föregående listorna till den. Skriv ut den baklänges.

4. `pop_back` och `push_front`

   Ta bort vartannat element ur den lista som fortfarande har 10 element genom att köra

   ```
   l.pop_back();
   l.push_front(l.pop_back());
   ```

   5 gånger.

5. `print_list`

   Skapa en global funktion

   ```
   void print_list(linked_list l);
   ```

   som du anropar med den halverade listan som argument. Listan ska skrivas ut.

   Observera att funktionen ska tillhöra det globala scopet (likt `main`) och att interfacet måste återges exakt — inga referenser eller konstanter. Du kan anropa listans inbyggda metoder för utskrifter väl inne i funktionen.

6. `merge`

   Skapa en global funktion

   ```
   linked_list merge(linked_list&, linked_list&);
   ```

   som du anropar med de två första listorna som argument. Funktionen ska kombinera de två listorna till en ny sorterad lista som sedan returneras. Listorna som matas in behöver inte vara intakta efter operationen. Du ska kunna lösa uppgiften med endast metoderna `front`, `is_empty`, `pop_front`, `push_back`. Pluspoäng om du även ser hur du kan nyttja `operator+=`.

7. Skapa och anropa en global funktion som kontrollerar att den returnerade listan från föregående uppgift är sorterad.

Kom ihåg att skriva ut listornas innehåll och andra väl valda detaljer, t.ex. listans storlek eller värdet i borttaget element där det är lämpligt **efter varje test**, samt en header som beskriver testet och samtidigt fungerar som avskiljare. Det ska alltså tydligt framgå i utdata **vad som testas**, **vad resultatet blir** och där det är tillämpligt även **varför**. Det går alldeles utmärkt att köra alla tester omedelbart efter varandra, utan paus, och utan någon sorts menysystem så länge du löser detta snyggt.

Krav

Följande krav framgår inte av API:t men ska också beaktas:

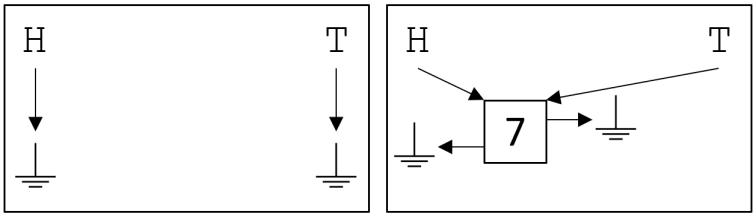
- Din specifikation och implementation ska uppdelas i lämpliga headerfiler och cpp-filer.
- Listan ska vara dubbellänkad.
- Listan ska ha nollbaserat index.
- Listan och dess noder får inte vid något tillfälle ha några dangling pointers eller minnesläckor som en följd av någon operation på listan. Städa efter dig och nollställ pekare.
- Alla metoder i API:t måste vara implementerade och testade. Det är OK att göra tillägg i form av ytterligare metoder.
- Kopieringskonstruktorn och tilldelningsoperatorn måste göra djupa kopior (*deep copy*).
- Tilldelningsoperatorn måste klara av självtilldelning (*self-assignment*).
- Endast en del av listans metoder testas genom de tidigare angivna testerna. Du måste därför komplettera dessa med egna tester.
- `using namespace std;` är inte tillåtet.
- Koden ska vara kommenterad till en rimlig grad. Förklara varför, inte vad.

5 Genomförande

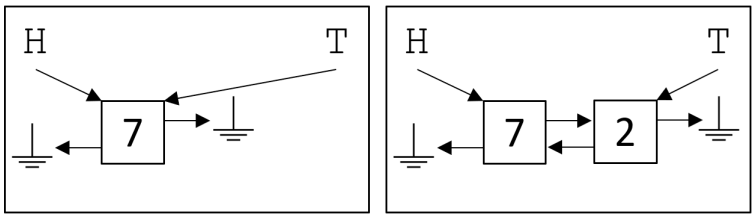
Att skapa en container från scratch är en stor uppgift. Du bör börja med att skapa de relevanta filerna du kommer behöva för din länkade lista. En headerfil för alla deklarationer som du fått ovan och en cpp-fil för att implementera dessa samt en cpp-fil för dina tester i main. Implementera först en konstruktor för listan så att du kan skapa en tom lista. Därefter kan du skapa nodens konstruktor och gå vidare till någon av metoderna som lägger till element, exempelvis `push_back`. Det är förstås också tacksamt att ha tillgång till `print` ur testsynpunkt så den bör skapas tidigt.

För varje funktion som modifierar listan måste du utgå ifrån de olika tillstånden listan kan befinna sig i och beakta hur det påverkar vad funktionen behöver

åstadkomma. Vi kan ta `push_back` som ett exempel. Om listan är tom innebär det att `push_back` skapar den första noden. Då måste både `head` och `tail` peka på den noden:



Om det istället redan finns ett element i listan så måste du länka ihop den nya noden med den tidigare svansen utan att tappa någon pekare längs vägen, samt flytta `tail`:



I bilderna ovan representeras `nullptr` av $\perp$. Siffrorna är värdena som lagras i respektive nod. Huvudet och svansen är `H` och `T`, respektive.

När du arbetar med länkade listor är det mycket nyttigt att rita upp de olika situationerna på papper. Ordningen du flyttar på pekarna är viktig för att du inte ska råka slarva bort en nod eller peka på något som inte finns (en dangling pointer).

Oroa dig inte över testerna i main just nu — skapa dina egna, enklare tester och laborera med metoderna så att du känner dig bekväm med att de fungerar.

Kom ihåg att återanvända din egna metoder i så stor utsträckning som möjligt så att du undviker upprepning. Detta är initialt rätt svårt då du kokar soppa på spik. Kom därför ihåg att då och då gå tillbaks till dina metoder för **refactoring**. Du kanske först senare inser att din `insert` kan använda `push_front` när den lägger till ett element i början av listan, istället för att jonglera 7 pekare. Sträva efter så lite upprepning som möjligt då det minskar risken för fel.

Forsätt göra små tillägg, testa ofta, och våga gå tillbaka och ändra i din kod. Det är nyckeln till denna uppgift.

FAQ

Q: *Jag vill köra size som en datamedlem istället, är det ok?*

A: Absolut, så länge du är införstådd med konsekvenserna det medför för alla metoder som arbetar med tillägg och borttagning.

Q: *Jag vill implementera listan som en generic container som håller typ T istället för doubles, är det ok?*

A: En smula överkurs, men jag tänker inte hindra dig. En ambitiös ansats för dig som vill ligga steget före, men beakta att du då gör det på eget bevåg utan särskild handledning för att uppnå detta.

Q: *Ska inte klassnamn alltid börja med stor bokstav?*

A: Generellt sett, ja. Vi gör ett undantag då `linked_list` är en containerklass likt `vector` och `string`. Se även: https://www.youtube.com/watch?v=jlOhMfqNQ-g.

6 Examination

Redovisa laborationen under ett av laborationstillfällena som ges under kursens gång genom att presentera funktionaliteten och förklara koden. När du fått klartecken från labbhandledaren att redovisningen är godkänd kan du lämna in i inlämningslådan på lärplattformen där en slutgiltlig bedömning utförs av examinator.

Din inlämning ska bestå av en **zip-fil** bestående av ett antal headerfiler och cpp-filer. Den fil som innehåller funktionen main bör heta `main.cpp`. Varje fil ska innehålla en header med åtminstone ditt namn och aktuellt datum.

6.1 Bedömning och återkoppling

Uppgiften bedöms med betygen *Godkänd (G)*, *Komplettering (Fx)*, och *Underkänd (U)*. Bedömningen baseras i huvudsak på huruvida fullständig funktionalitet uppnåtts och om du under den muntliga delen av examinationen kunnat förklara innebörden av din kod.

Återkoppling erhåller du i första hand muntligt under redovisningen. Normalt lämnas ingen skriftlig återkoppling vid godkänt resultat för denna uppgift.

Referenser

[1] S. B. Lippman, *C++ primer*, 5th ed. Upper Saddle River, N.J.: Addison-Wesley, 2013.