TDDD86 – Laboration #3

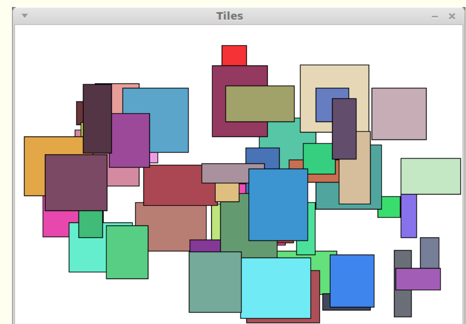
11 november 2020

I den här tvådelade uppgiften får du öva på att implementera och använda datastrukturer baserade på utökningsbara arrayer och länkade listor. Filerna du behöver för att komma igång finns som `labb3.tar.gz` på kurshemsidan.

Redovisning: Efter att du redovisat muntligt, gör en `git commit -m ''TDDD86 Lab 3 redovisning''` och en `git push`. Se till att filerna **TileList.h**, **TileList.cpp**, **Tour.h** och **Tour.cpp**är med. Informera sedan din assistant genom att meddela honom/henne (använd @assistant_namn) i privata kanalen i labbgrupps teamet.

Del A: Tiles

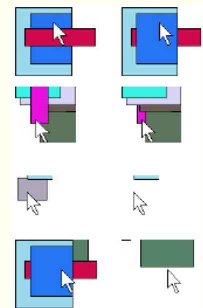
I del A av den här labben kommer du att skriva logiken för ett grafiskt program som låter användaren klicka på rektangulära plattor. Du kommer att skriva en klass som lagrar en lista av plattor (*tiles*) som använder en array/ett fält för den interna lagringen. Programmet är grafiskt, men du behöver inte direkt rita någon grafik själv — all grafikkod är redan skriven och du behöver bara anropa den.



Huvudprogrammet `tilemain.cpp` använder klassen `mainwindow` för att skapa ett grafiskt fönster på skärmen och visa en lista av plattor. (Initialt visas 50 plattor, men du kan trycka på `'N'` för att lägga till fler.) Varje platta är en struct av den tillhandahållna typen `Tile`. Varje plattas position, storlek och färg genereras slumpmässigt av huvudprogrammet. Tanken är nu att en lista av alla plattor ska lagras och underhållas av din kod i `TileList`-klassen du kommer att skriva.

Lägg märke till att, i skärmbilden till höger, överlappar vissa plattor och täcker delvis samma pixlar (x, y) i fönstret. I sådana fall är plattan som skapats senare “ovanpå” tidigare plattor och kan täcka dem partiellt på skärmen. I uppgiften bör du tänka på listan av plattor som att den har en ordning, där plattor som lagras tidigare i listan är närmre “botten” och där sådana som lagras senare i listan är närmre “toppen”. Detta kallas ibland för en *z-ordning*.

Det grafiska användargränssnittet visar plattorna och låter användaren klicka på dem, med följande resultat:



- Om användaren klickar på **vänster musknapp** när musen pekar på en platta flyttas den plattan till toppen av z-ordningen (slutet av listan).
- Om användaren klickar på **vänster musknapp** samtidigt som hen håller ned **Shift** när musen pekar på en platta flyttas den plattan till botten av z-ordningen (början av listan).
- Om användaren klickar på **höger musknapp** när musen pekar på en platta tas den plattan bort från listan och försvinner från skärmen.
- Om användaren klickar på **höger musknapp** samtidigt som hen håller ned **Shift** tas alla plattor som täcker den pixeln bort från listan och försvinner från skärmen.
- Om användaren trycker på **tangentbordets N-knapp** skapas en slumpmässigt positionerad platta och läggs till på skärmen.

Om användaren klickar på en pixel som täcks av fler än en platta används den som är längst upp. (Förutom vid ett shiftat högerklick, då alla plattor som täcker pixeln tas bort, inte bara den översta.)

Notera att **din kod inte direkt behöver detektera musklick eller tangenttryckningar**. Den tillhandahålla koden upptäcker klick/tryckningar och reagerar genom att anropa diverse metoder i ditt `TileList`-objekt enligt följande beskrivning.

Del A, implementationsdetaljer:
Din `TileList`-klass representerar internt en lista av plattor med en **array av `Tile`-struktar**. Det är ditt jobb att utöka arrayen om det behövs när den blir för full för att kunna lagra en till platta och också att flytta element till höger/vänster vid behov för att omarrangera plattorna. Skriv följande publika medlemmar. **Antag att all indata är giltigt**.

`TileList()`	I denna konstruktor initialiserar du en ny tom lista av plattor med kapacitet 10.
`~TileList()`	I denna destruktor ska du frigöra allt dynamiskt allokerat minne som används av din lista.
`addTile(tile)`	I denna metod ska du lägga till den givna plattan till slutet (toppen) av listan. $\mathcal{O}(1)$ (amorterad komplexitet).
`drawAll(screen)`	I denna metod får du en pekare till ett `QGraphicsScene`-objekt och du ska rita alla plattor i din lista på det objektet i rätt ordning genom att använda respektive plattas `draw`-funktion. Plattor som ligger tidigare i listan ska uppträda "under" plattor som ligger senare i listan. $\mathcal{O}(N)$.
`indexOfTopTile(x, y)`	I denna metod ska du returnera det 0-baserade indexet för den översta (sista) plattan i listan som täcker en given (x, y)-position. Om ingen platta täcker positionen, returnera -1. $\mathcal{O}(N)$.
`raise(x, y)`	Anropas av GUI:t när användaren vänsterklickar på de givna x/y-koordinaterna. Om dessa koordinater täcks av någon platta ska du flytta den översta (sista) av dessa plattor till toppen (sist) av listan. Flytta om element om det behövs. $\mathcal{O}(N)$.
`lower(x, y)`	Anropas av GUI:t när användaren shift-vänsterklickar på de givna x/y-koordinaterna. Om dessa koordinater täcks av någon platta ska du flytta den översta (sista) av dessa plattor till botten (först) av listan. Flytta om element om det behövs. $\mathcal{O}(N)$.
`remove(x, y)`	Anropas av GUI:t när användaren högerklickar på de givna x/y-koordinaterna. Om dessa koordinater täcks av någon platta ska du ta bort den översta (sista) av dessa plattor från listan. Flytta om element om det behövs. $\mathcal{O}(N)$.
`removeAll(x, y)`	Anropas av GUI:t när användaren shift-högerklickar på de givna x/y-koordinaterna. Om dessa koordinater täcks av någon platta ska du ta bort alla sådana plattor från listan. Flytta om element om det behövs. $\mathcal{O}(N^2)$.

Varje `Tile`-struktur i din lista har följande publika medlemmar. Se `Tile.h` för mer detaljer.

`x, y, width, height`	Övre/vänstra (x, y)-koordinaten för plattan och dess bredd och höjd i pixlar.
`r, g, b`	Plattans färg, specificerad som RGB-värde.
`contains(x, y)`	Returnerar `true` om denna platta täcker den givna (x, y)-positionen.
`draw(screen)`	Ritar plattan på det givna `QGraphicsScene`-objektet.
`toString()`	Returnerar en textrepresentation av plattan, vilket kan vara användbart för felsökning.

I filen `TileList.h` finns en påbörjad version av headerfilen som deklarerar ovanstående medlemmar. Du kommer att behöva modifiera denna fil för att slutföra arbetet. I synnerhet behöver du göra följande:

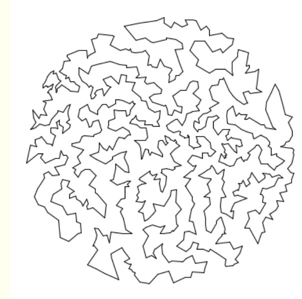
- **Lägg till kommentarer** i `TileList.h` och `TileList.cpp`. (Kommentarer till metodhuvuden ska placeras i `.h`-filen. Implementationerna av dessa medlemmar i `.cpp`-filen behöver inte några kommentarer i metodhuvudet.)
- **Deklarera nödvändiga privata medlemmar** i `TileList.h`, så som privata instansvariabler vilka behövs för att implementera beteendet. Din inre datastruktur *måste* vara en array/ett fält av `Tiles`, använd inte någon annan datastruktur.
- **Lägg till** `const` där det är lämpligt till de publika medlemmar som inte modifierar listans tillstånd.
- **Implementera kropparna** för alla medlemsfunktioner och konstruktorer i `TileList.cpp`. Flera operationer är likartade. Undvik redundans i din kod genom att skapa ytterligare hjäpfunktioner i din `TileList`. (Deklarera dem `private` så att inte utomstående kod kan anropa dem.)

Vi föreslår att du enbart kodar de basala operationer först, som `addTile` och `drawAll`. Skriv sedan `raise` och sedan resten. För att lista ut om en platta täcker en given x/y-pixel anropar du plattans `contains`-metod. Skjut upp att bry dig om att utöka arrayen genom att ändra huvudprogrammet till att använda färre rektanglar (säg 5) till att börja med.

Del B: TSP

Givet N punkter i planet är målet för en handelsresande att besöka alla punkter (och komma hem igen) samtidigt som den totala längden av resan är så kort som möjligt. Implementera två giriga heuristiker för att hitta bra (men inte optimala) lösningar till handelsresandeproblemet (*Traveling Salesman Problem, TSP*).

(a) 1000 punkter.



(b) Optimal tur.

TSP är ett viktigt problem, inte bara för att det finns så många handelsresanden som vill minimera sitt resande, utan snarare för den stora mängden tillämpningsområden som fordonsruttning, kretskortsborrning, VLSI-design, robotstyrning, röntgenkristallografi, schemaläggning och beräkningsbiologi.

TSP är ett omryktat svårt kombinatoriskt optimeringsproblem. I princip kan man enumerera alla möjliga turer och välja den kortaste — i praktiken är antalet möjliga turer så stort (ungefär $N!$) att detta inte är görbart. För stora N känner ingen till någon effektiv metod för att hitta den kortaste möjliga turen för en given mängd punkter. Däremot har många metoder studerats som verkar fungera väl i praktiken, även om de inte är garanterade att producera den bästa möjliga turen. Sådana metoder kallas *heuristiker*. I den här uppgiften ska du implementera insättningsheuristikerna **nearest neighbour** och **smallest increase** som bygger upp en tur inkrementellt. Börja med en en-punktstur (från första punkten tillbaka till sig själv) och iterera följande process till det inte finns några punkter kvar.

- **Nearest neighbour**: Läs in nästa punkt och lägg till den till turen *efter* den punkt vilken den är närmast. (Använd den första sådana punkten om det finns flera punkter som är närmast.)
- **Smallest increase**: Läs in nästa punkt och lägg till den till turen *efter* den punkt där den resulterar i minsta möjliga ökning av turlängden. (Använd den första sådana punkten om det finns flera punkter med den egenskapen.)

Du har tillgång till en datatyp för punkter i planet, `Point`, med följande gränssnitt.

```
class Point (2D point data type)
-----------------------------------------------------------------------------
       Point(double x, double y)                  // create the point (x, y)
string toString()                                 // return string representation
  void draw(QGraphicsScene *scene)                // draw point on scene
  void drawTo(Point that, QGraphicsScene *scene)  // draw line segment between
                                                  // the two points
double distanceTo(Point that)                     // return Euclidean distance
                                                  // between the two points
```

Din uppgift är att skapa en datatyp, `Tour`, som representerar en sekvens av punkter besökta i en TSP-tur. Representera turen som en *cirkulär länkad lista* av noder, en för varje punkt. Till din hjälp finns en färdig nod-datatyp, `Node`, där varje nod har en `Point` och en pekare till nästa `Node` i turen. Dessutom har du tillgång till huvudprogrammet `tsp.cpp` som använder din `Tour`-klass för att hitta bra lösningar till givna instanser av TSP-problemet. Bland de givna filerna finns också många testfiler att prova med och förväntat utdata för vissa av dessa.

Del B, implementationsdetaljer:
I den här deluppgiften specificerar vi exakt vilka datamedlemmar du får ha i `Tour`-klassen. Du måste ha exakt följande datamedlem, du får inte ha några andra.

- en pekare till första noden i den cirkulära länkade listan

Du måste använda vår `Node`-struktur och din länkade lista måste ha följande publika medlemmar. Se `Tour.h` för mer detaljer.

```
class Tour
--------------------------------------------------------------------------------
       Tour()                         // create an empty tour
       ~Tour()                        // free all memory used by list nodes
  void show()                         // print the tour to standard output
  void draw(QGraphicsscene *scene)    // draw the tour on scene
   int size()                         // number of points on tour
double distance()                     // return the total distance of the tour
  void insertNearest(Point p)         // insert p using nearest neighbor heuristic
  void insertSmallest(Point p)        // insert p using smallest increase heuristic
```

Observera att det här är tänkt att vara en övning i att manipulera länkade listor. Det betyder att du måste använda vår `Node`-struktur och att du inte får modifiera den. Du får inte heller skapa några arrayer, `vector`s, stackar, köer, mängder, avbildningar, andra STL-containrar eller andra datastrukturer; du måste använda länkade noder. Precis som i del A vill vi att du lägger till kommentarer, deklarerar de privata medlemmar som behövs och applicerar `const` där så är tillämpligt. Lämplig arbetsgång:

1. Studera den givna koden.
2. I felsökningssyfte kan du skapa en konstruktor `Tour(Point a, Point b, Point c, Point d)` som tar fyra punkter och skapar en cirkulär länkad lista med dessa fyra punkter i. Skapa först fyra noder och tilldela en punkt till varje. Länka sedan samman noderna i en cirkel.
3. Implementera metoden `show`. Den ska traversera varje nod i den cirkulära länkade listan med början i den första noden och skriva ut varje punkt. Den här metoden kräver bara några rader kod men det är ändå viktigt att tänka noga på dem eftersom att felsöka länkade listor kan vara svårt och frustrerande. Börja med att bara skriva ut den första punkten. I cirkulära länkade listor pekar den sista noden tillbaka på den första, så se upp för oändliga loopar.
   Testa metoden genom att ändra i huvudprogrammet så att det definierar fyra punkter, skapar en ny tur av dessa fyra punkter och anropar turens `show`-metod. Nedan följer ett förslag på hur det kan se ut.
   ```
   // define 4 points forming a square
   Point p(100.0, 100.0);
   Point q(500.0, 100.0);
   Point r(500.0, 500.0);
   Point s(100.0, 500.0);

   // Set up a Tour with those four points
   // The constructor should link p->q->r->s->p
   Tour squareTour(p, q, r, s);

   // Output the Tour
   squareTour.show();
   ```
   Om din metod fungerar korrekt får du följande utdata.
   ```
   (100.0, 100.0)
   (500.0, 100.0)
   (500.0, 500.0)
   (100.0, 500.0)
   ```
   Testa din metod `show()` på turer med 0, 1 eller 2 punkter och kontrollera att den fortfarande fungerar. Du kan skapa sådana instanser genom att modifiera 4-nodskonstruktorn till att bara länka samma 0, 1 eller 2 av de givna punkterna. (Om du inte tilldelar instansvariabeln som pekar ut den första noden har du en tom tur. Om du sätter instansvariabeln att peka på någon nod och länkar den tillbaka till sig själv har du en tur som innehåller en punkt.)
4. Implementera `size()`. Den blir väldigt lik `show()`.
5. Implementera `distance()`. Den blir väldigt lik `show()`, så när som på att du måste anropa `distanceTo()` i punktdatatypen. Om du lägger till ett anrop till `distance()` i den kvadratiska turen du skapade ovan bör resultatet bli `1600.0`.
6. Implementera `draw()`. Den är också väldigt lik `show()`, så när som på att du måste anropa `drawTo()` i punktdatatypen.
7. Implementera `insertNearest()`. För att avgöra vilken nod du ska sätta in `p` efter behöver du beräkna det euklidiska avståndet mellan varje punkt i turen och `p` genom att traversera den cirkulära länkade listan. Lagra hela tiden noden som innehåller den hittills närmsta punkten och dess avstånd till `p`. Efter att du hittat den närmsta noden skapar du en nod som innehåller `p` och sätter in den *efter* den närmsta noden. Det betyder att du måste ändra på `next`-pekaren i både den nyligen skapade noden och den

närmsta noden. Bland de givna filerna finns `tsp10-nearest.ans` som innehåller det förväntade resultatet, en tur med längd `1566.1363`, för 10-punktersproblemet `tsp10.txt`. Notera att den optimala turen har längd `1552.9612`, så den här heuristiken ger inte den bästa turen i allmänhet.

8. Nu borde det vara relativt enkelt att skriva `insertSmallest()`. Den enda skillnaden är att du ska sätta in punkten `p` där den resulterar i minsta möjliga ökning av turens totala längd. Som kontrollpunkt finns filen `tsp10-smallest.ans` som innehåller det förväntade resultatet, en tur med längd `1655.7462`, för 10-punktersproblemet `tsp10.txt`. I det här fallet är **smallest insertion**-heuristiken faktiskt sämre än **nearest insertion**-heuristiken (trots att detta inte är typiskt).

Möjliga utökningar

E3 — en bättre heuristik (3 poäng):
Lägg märke till att en tur som har delsträckor som korsar varandra kan transformeras till en kortare tur utan korsningar. För att göra detta kan det vara nödvändigt att lägga till fler metoder i `Tour`. Skapa en ny branch som heter E3 och implementera denna strategi som en ny publik medlem i `Tour`. Efter att du redovisat muntligt, gör en `git commit -m ``TDDD86 E3 redovisning''` och en `git push`. Se till att filerna **Tour.h** och **Tour.cpp**är med. Informera sedan din assistant genom att meddela honom/henne (använd @assistant_namn) i privata kanalen i labbgrupps teamet.

E4 — egen heuristik (3 poäng):
Implementera en egen heuristik som ger, i 60 sek eller mindre, minst 1% kortare väg än insert-smallest på tsp1000. För att göra detta kan det vara nödvändigt att lägga till fler metoder i `Tour`. Utdataformatet måste dock vara samma som i det befintliga programmet. Skapa en ny branch som heter E4. Efter att du redovisat muntligt, gör en `git commit -m ``TDDD86 E4 redovisning''` och en `git push`. Se till att filerna **Tour.h** och **Tour.cpp**är med. Informera sedan din assistant genom att meddela honom/henne (använd @assistant_namn) i privata kanalen i labbgrupps teamet.

E5 — tävling (2 poäng till vinnaren):
Laget med det vinnare programmet tilldelas 2 poäng. Programmet måste skickas in innan den deadline som anges på kurshemsidan. Juryn kommer att plocka 5 banor bland dem som kommer med i uppgiften. För varje sekund över 60sek (på maskinerna i SU-salen) per bana får programmet som straff 2% extra i distans för den banan. Om en bana tar mer än 120 sek diskvalificeras programmet. Vinnaren blir det icke-diskvalificerade programmet som ger minsta summan. I övrigt gäller samma begränsningar som i E4. Skapa en ny branch som heter E5. Efter att du redovisat muntligt, gör en `git commit -m ``TDDD86 E5 redovisning''` och en `git push`. Informera sedan din assistant genom att meddela honom/henne (använd @assistant_namn) i privata kanalen i labbgrupps teamet.